(12) United States Patent
Porterfield

(10) **Patent No.:** **US 6,418,523 B2**
(45) **Date of Patent:** **Jul. 9, 2002**

(54) **APPARATUS COMPRISING A TRANSLATION LOOKASIDE BUFFER FOR GRAPHICS ADDRESS REMAPPING OF VIRTUAL ADDRESSES**

(75) Inventor: **A. Kent Porterfield**, New Brighton, MN (US)

(73) Assignee: **Micron Electronics, Inc.**, Nampa, ID (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/865,653**

(22) Filed: **May 24, 2001**

Related U.S. Application Data

(62) Division of application No. 08/882,054, filed on Jun. 25, 1997, now Pat. No. 6,249,853.

(51) **Int. Cl.**^{7} **G06F 12/10**

(52) **U.S. Cl.** **711/207**; 345/568

(58) **Field of Search** 711/202, 203, 711/206, 207, 208; 345/568

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,016,545	A	4/1977	Lipovski
4,507,730	A	3/1985	Johnson et al.
4,937,734	A	6/1990	Bechtolsheim
4,969,122	A	11/1990	Jensen
5,121,487	A	6/1992	Bechtolsheim
5,133,058	A	7/1992	Jensen
5,155,816	A	10/1992	Kohn
5,222,222	A	6/1993	Mehring et al.
5,263,142	A	11/1993	Watkins et al.
5,265,213	A	11/1993	Weiser et al.
5,265,227	A	11/1993	Kohn et al.
5,265,236	A	11/1993	Mehring et al.
5,305,444	A	4/1994	Becker et al.
5,313,577	A	5/1994	Meinerth et al.
5,315,696	A	5/1994	Case et al.
5,315,698	A	5/1994	Case et al.
5,321,806	A	6/1994	Meinerth et al.
5,321,807	A	6/1994	Mumford
5,321,836	A	6/1994	Crawford et al.
5,361,340	A	11/1994	Kelly et al.
5,392,393	A	2/1995	Deering
5,396,614	A	3/1995	Khalidi et al.
5,408,605	A	4/1995	Deering

(List continued on next page.)

OTHER PUBLICATIONS

Accelerated Graphics Port Interface Specification. Revision 1.0 Intel Corporation. Jul. 31, 1996. 81 pgs.

Intel Advance information "INTEL 440LX AGPSET:82443LX PCI A.G.P. Controller (PAC)" Aug. 97, 139 pp.

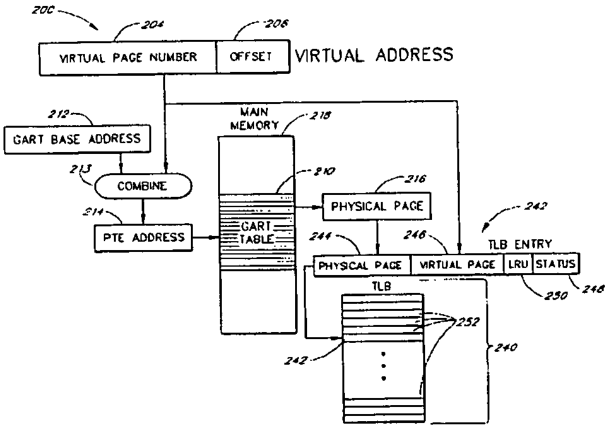
LSI Logic L64852 Mbus–to–Sbus Controller (M2S) Technical Manual. LSI Logic Corporation (1993). 73 pp.

Primary Examiner—Kevin Verbrugge
(74) *Attorney, Agent, or Firm*—Knobbe, Martens, Olson & Bear LLP

(57) ABSTRACT

A modular architecture for storing, addressing and retrieving graphics data from main memory instead of expensive local frame buffer memory. A graphic address remapping table (GART), defined in software, is used to remap virtual addresses falling within a selected range, the GART range, to non-contiguous pages in main memory. Virtual address not within the selected range are passed without modification. The GART includes page table entries (PTEs) having translation information to remap virtual addresses falling within the GART range to their corresponding physical addresses. The GART PTEs are of configurable length enabling optimization of GART size and the use of feature bits, such as status indicators, defined by software. The GART is implemented during system boot up by configuration registers. Similarly, the PTEs are configured using mask registers. The GART may be used in conjunction with a translation lookaside buffer (TLB) to improve address remapping performance.

18 Claims, 13 Drawing Sheets



U.S. PATENT DOCUMENTS

5,426,750	A	6/1995	Becker et al.
5,440,682	A	8/1995	Deering
5,446,854	A	8/1995	Khalidi et al.
5,465,337	A	11/1995	Kong
5,479,627	A	12/1995	Khalidi et al.
5,491,806	A	2/1996	Horstmann et al.
5,500,948	A	3/1996	Hinton et al.
5,524,233	A	6/1996	Milburn et al.
5,542,062	A	7/1996	Taylor et al.
5,546,555	A	8/1996	Horstmann et al.
5,548,739	A	8/1996	Yung
5,553,023	A	9/1996	Lau et al.
5,584,014	A	12/1996	Nayfeh et al.
5,586,283	A	12/1996	Lopez-Aguado et al.
5,664,161	A	9/1997	Fukushima et al.
5,675,750	A	10/1997	Matsumoto et al.
5,737,765	A	4/1998	Shigeeda
5,778,197	A	7/1998	Dunham
5,815,167	A	9/1998	Muthal et al.
5,845,327	A	12/1998	Rickard et al.
5,854,637	A	12/1998	Sturgess
5,861,893	A	1/1999	Sturgess
5,889,970	A	3/1999	Horan et al.
5,892,964	A	4/1999	Horan et al.
5,909,559	A	6/1999	So
5,911,051	A	6/1999	Carson et al.

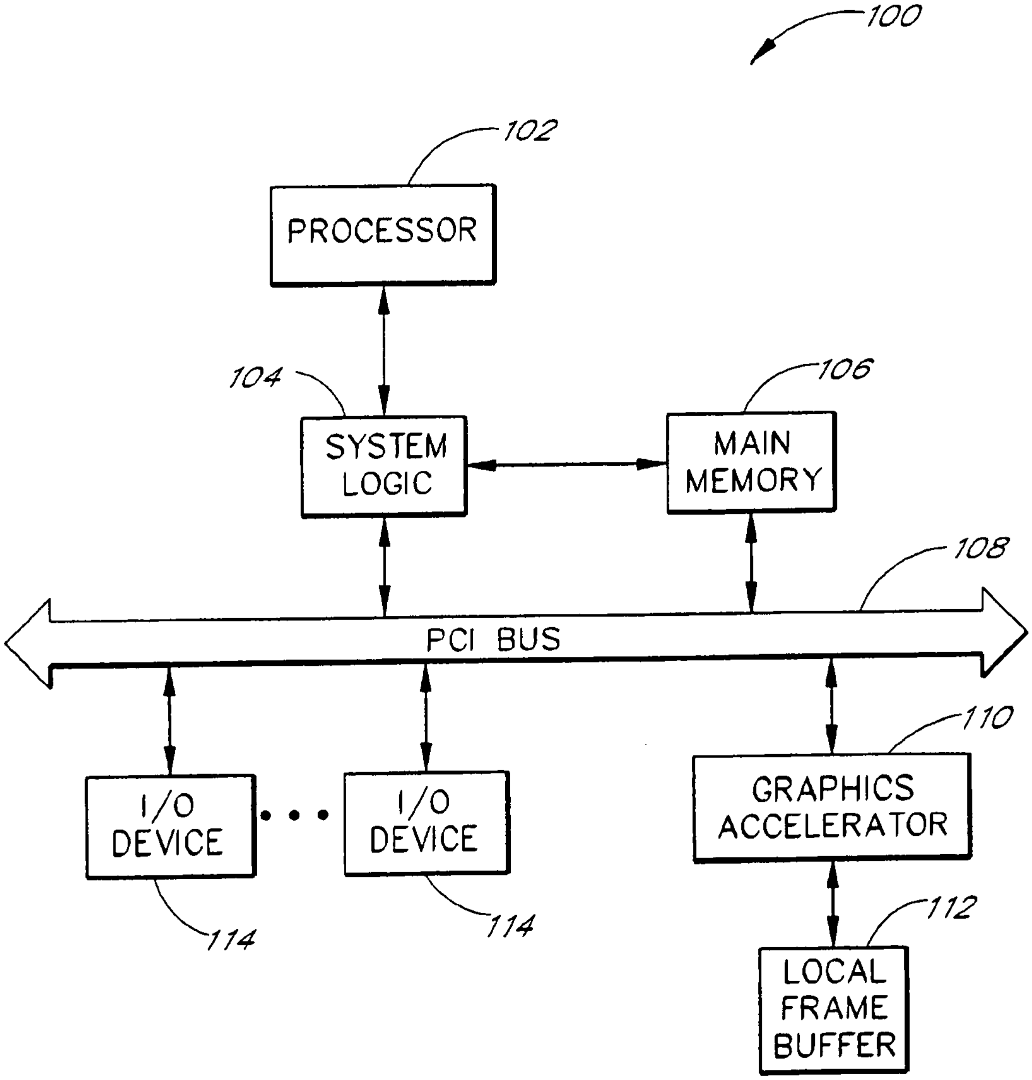


FIG. 1
(PRIOR ART)

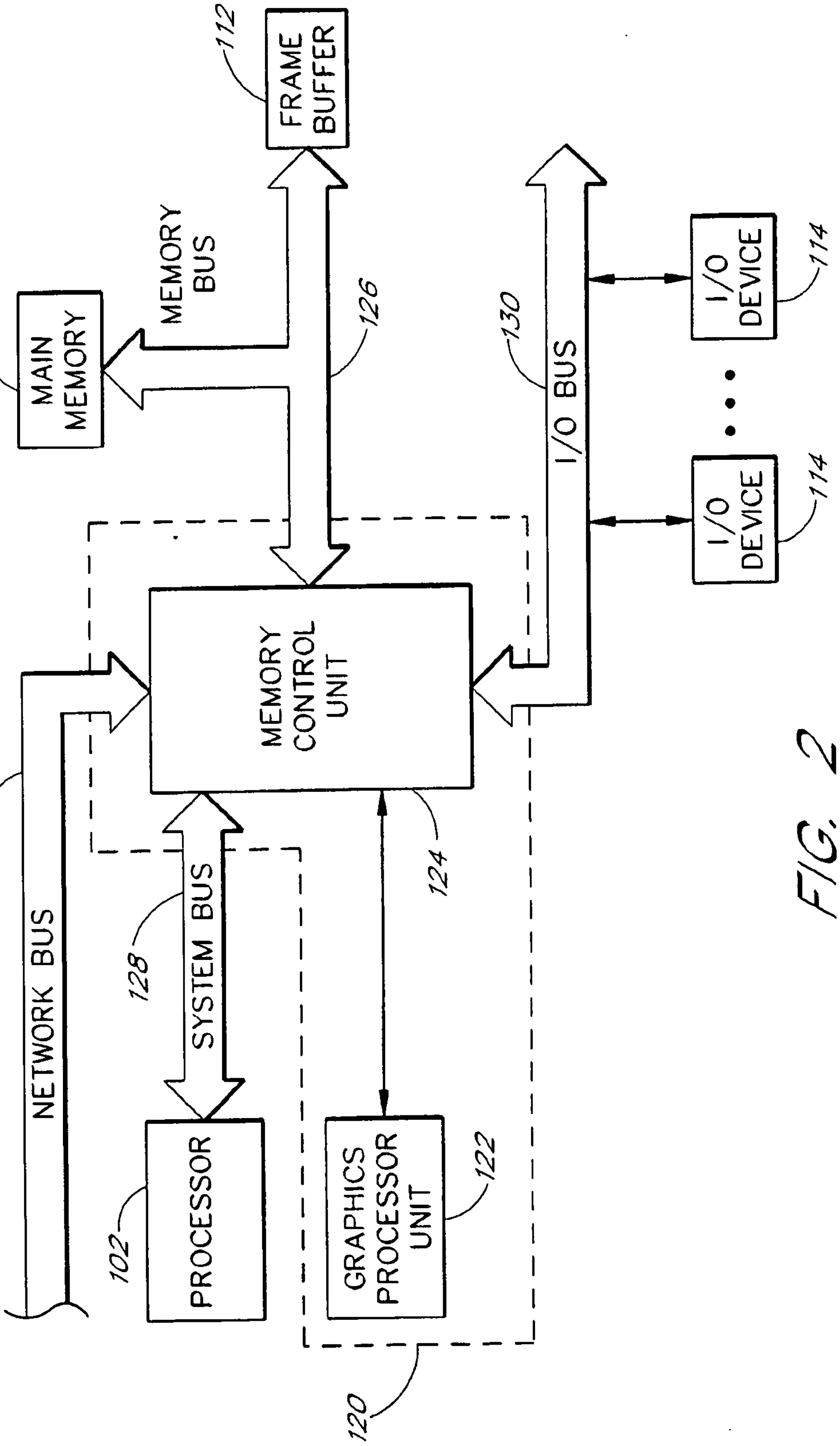


FIG. 2
(PRIOR ART)

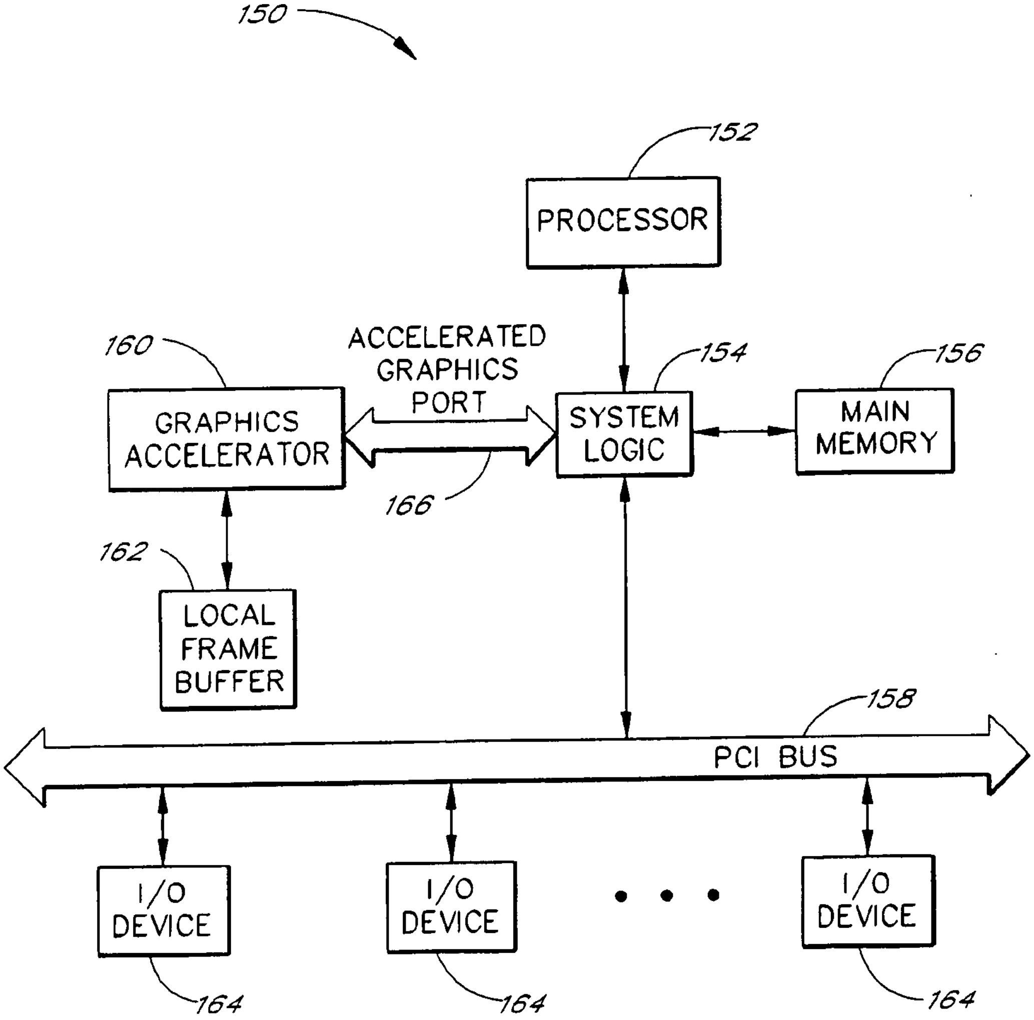

150
152
PROCESSOR
160
ACCELERATED
GRAPHICS
PORT
154
156
GRAPHICS
ACCELERATOR
SYSTEM
LOGIC
MAIN
MEMORY
166
162
LOCAL
FRAME
BUFFER
158
PCI BUS
I/O
DEVICE
I/O
DEVICE
• • •
I/O
DEVICE
164
164
164

FIG. 3

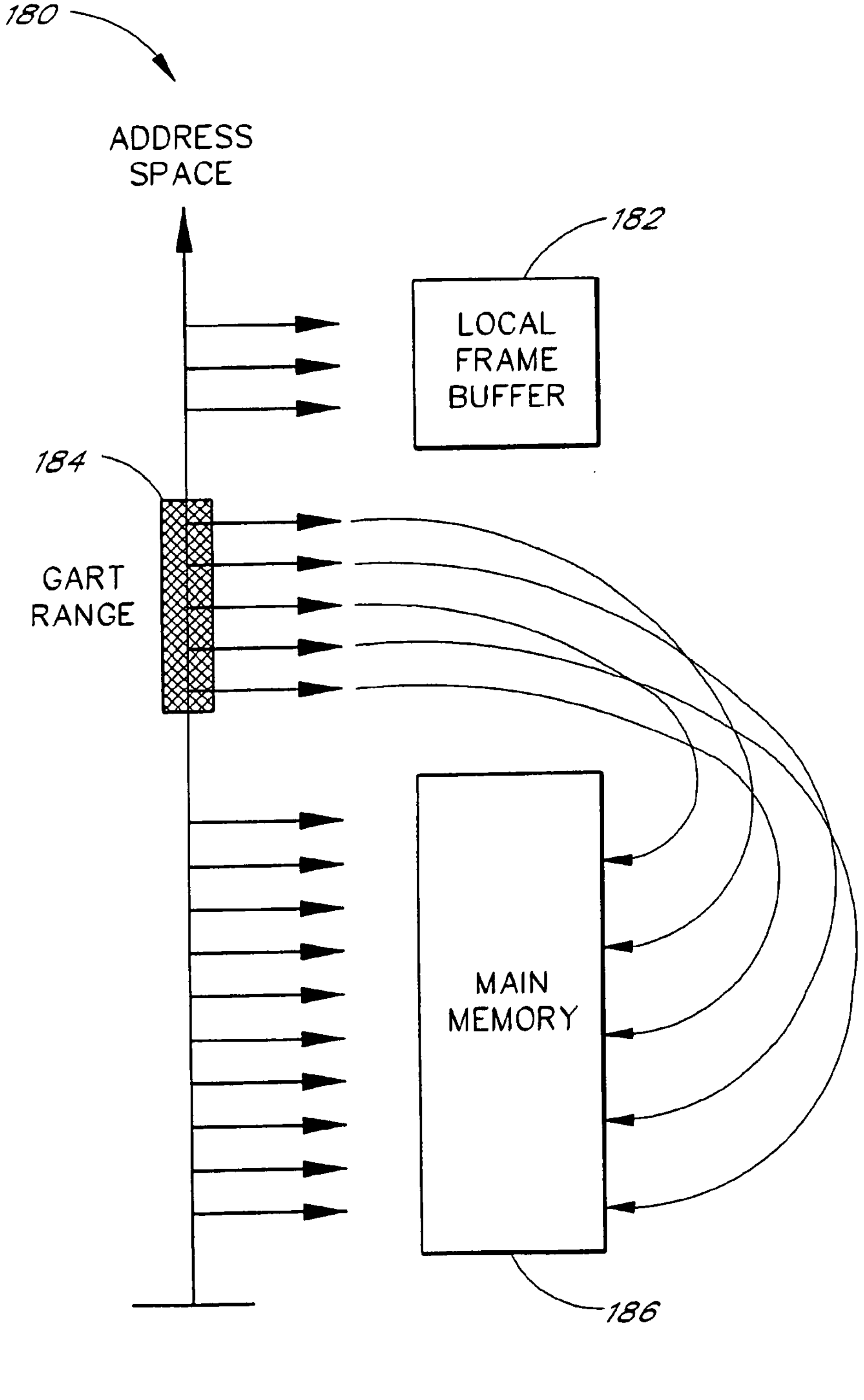

180
ADDRESS SPACE
182
LOCAL FRAME BUFFER
184
GART RANGE
MAIN MEMORY
186

FIG. 4

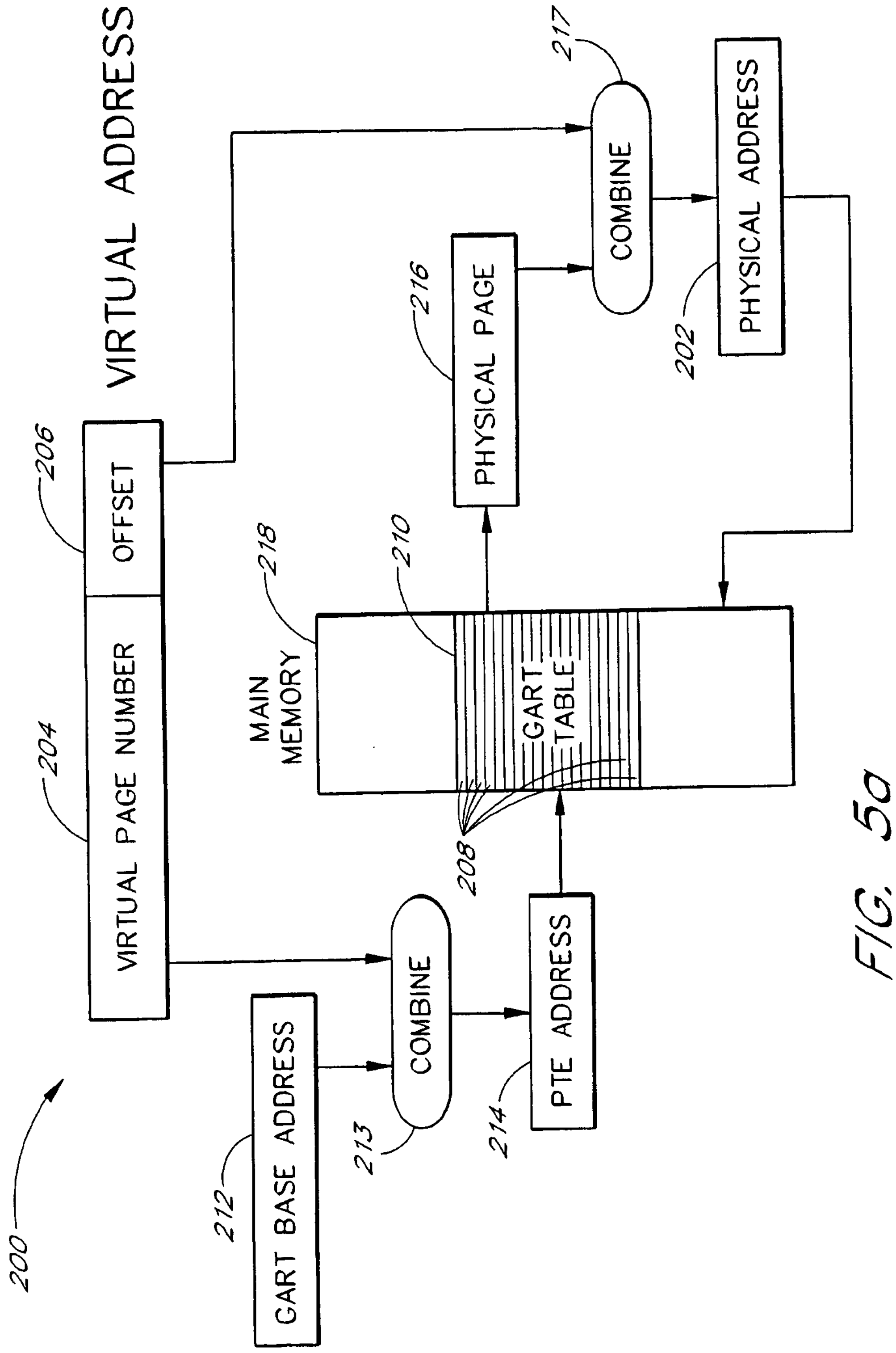

VIRTUAL ADDRESS
200
204
VIRTUAL PAGE NUMBER
206
OFFSET
212
GART BASE ADDRESS
213
COMBINE
214
PTE ADDRESS
208
218
MAIN MEMORY
210
GART TABLE
216
PHYSICAL PAGE
217
COMBINE
202
PHYSICAL ADDRESS

FIG. 5a

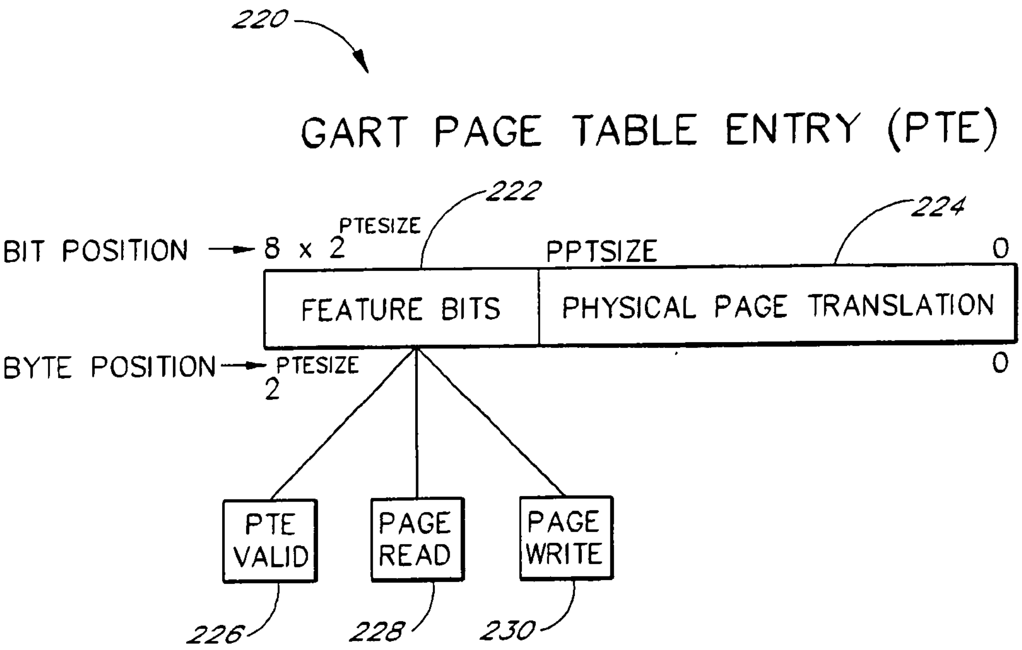

220
GART PAGE TABLE ENTRY (PTE)
222
224
BIT POSITION
8 x 2^PTESIZE
PPTSIZE
0
FEATURE BITS
PHYSICAL PAGE TRANSLATION
BYTE POSITION
2^PTESIZE
0
PTE VALID
PAGE READ
PAGE WRITE
226
228
230

FIG. 5b

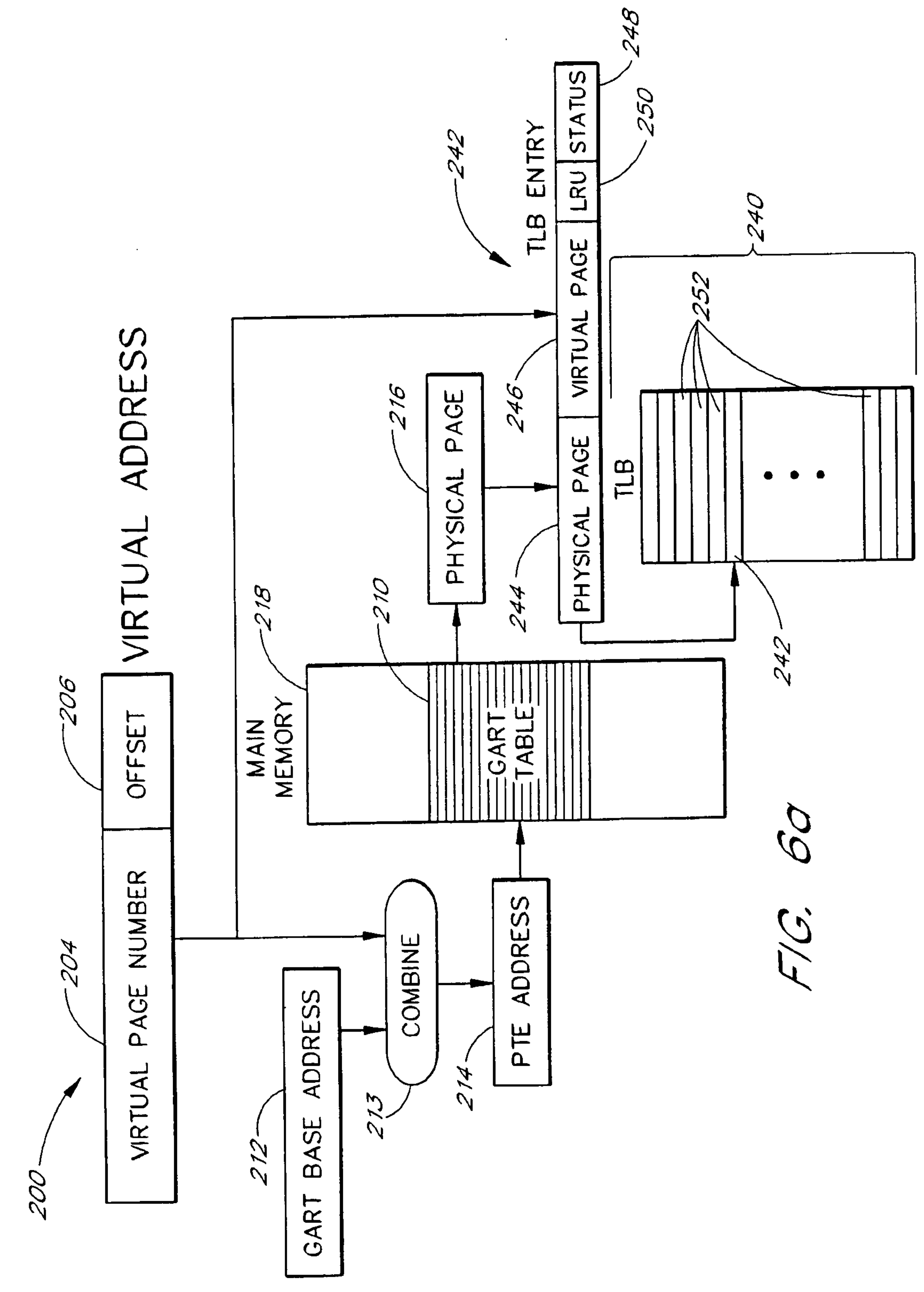

200
VIRTUAL ADDRESS
VIRTUAL PAGE NUMBER
204
OFFSET
206
GART BASE ADDRESS
212
COMBINE
213
PTE ADDRESS
214
MAIN MEMORY
218
GART TABLE
210
PHYSICAL PAGE
216
PHYSICAL PAGE
244
VIRTUAL PAGE
246
LRU
250
STATUS
248
TLB ENTRY
242
TLB
240
252
242

FIG. 6a

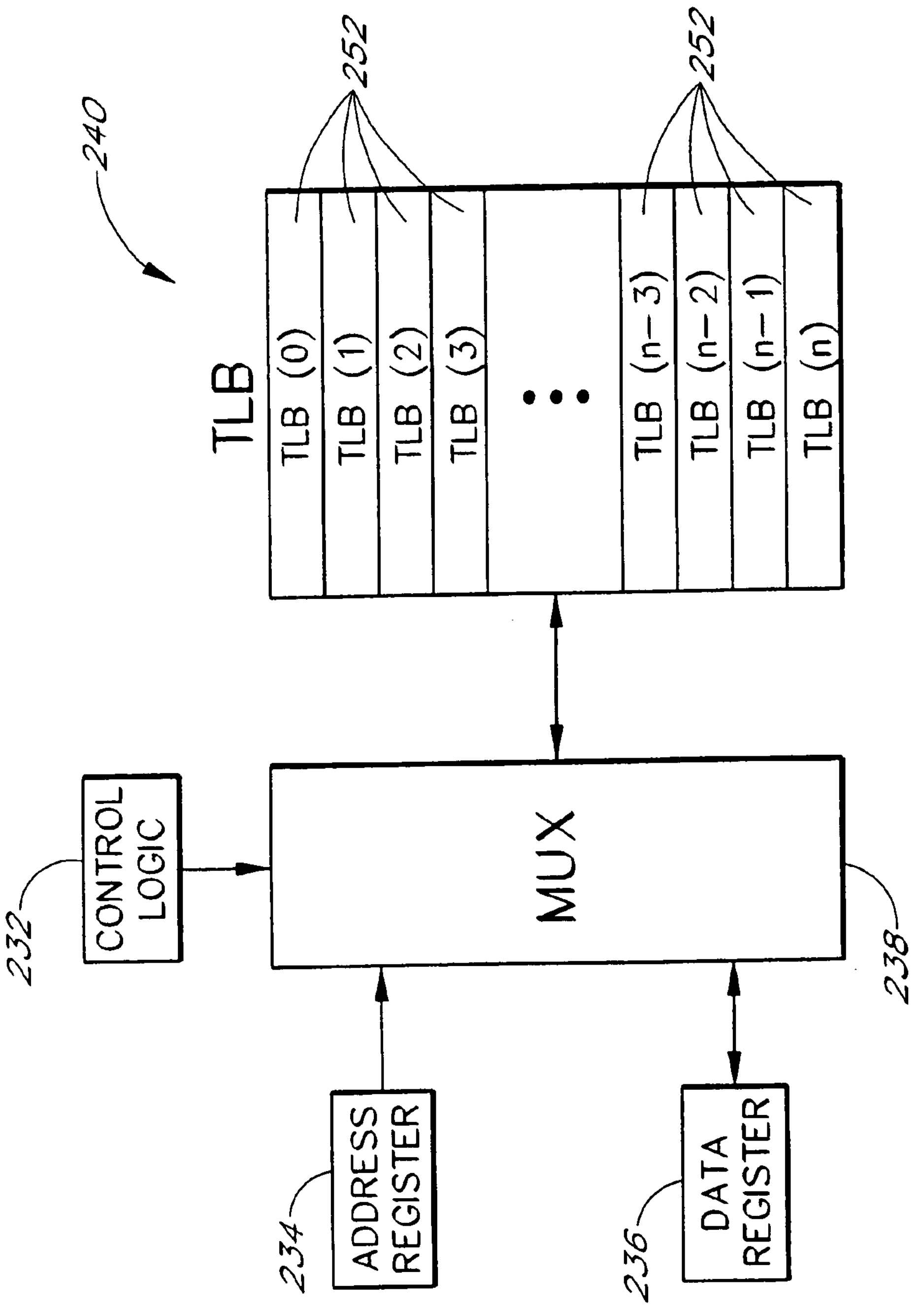

240
252
252
TLB
TLB (0)
TLB (1)
TLB (2)
TLB (3)
TLB (n−3)
TLB (n−2)
TLB (n−1)
TLB (n)
MUX
CONTROL LOGIC
232
ADDRESS REGISTER
234
DATA REGISTER
236
238

FIG. 6b

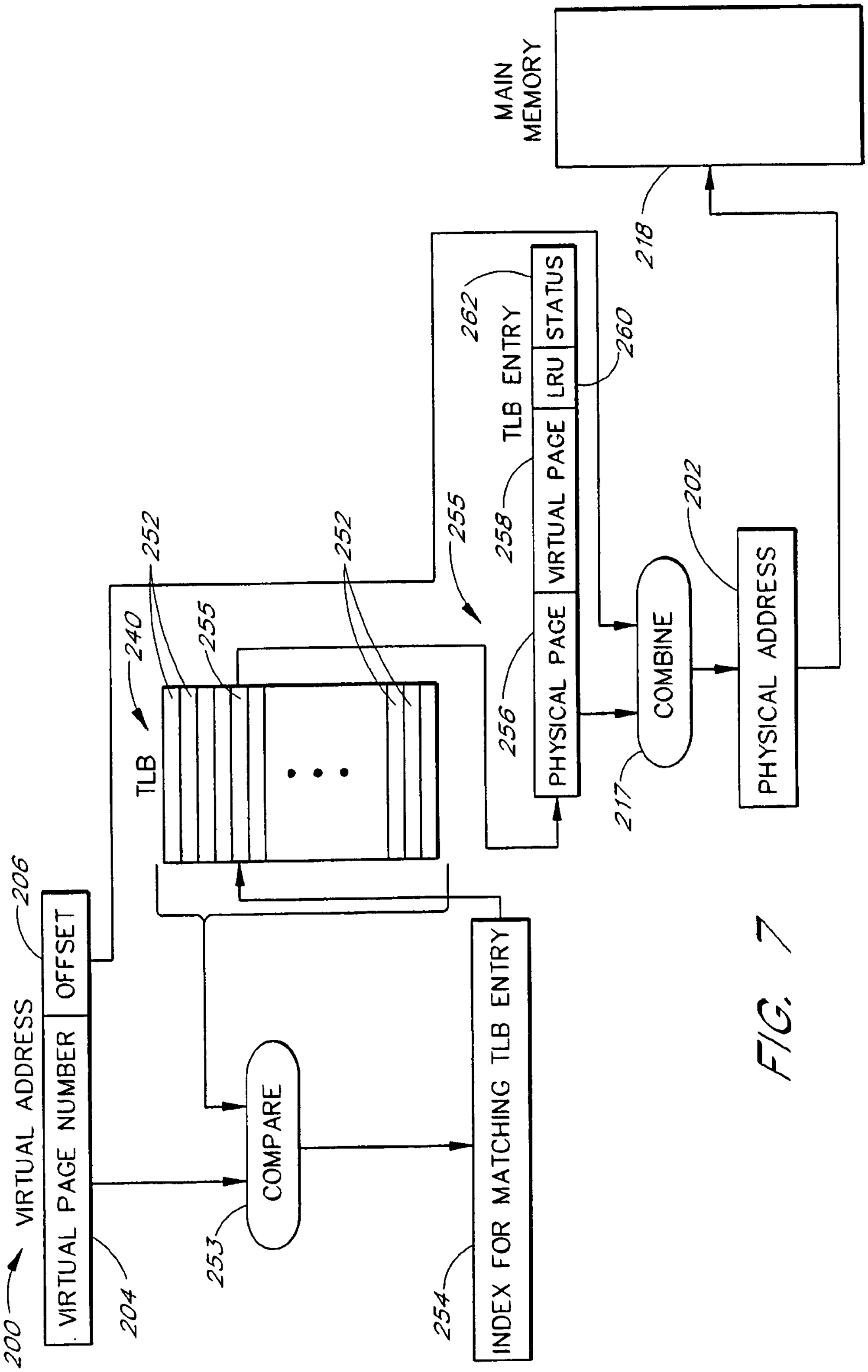

200
VIRTUAL ADDRESS
VIRTUAL PAGE NUMBER
OFFSET
204
206
COMPARE
253
INDEX FOR MATCHING TLB ENTRY
254
TLB
240
252
255
252
255
TLB ENTRY
PHYSICAL PAGE
VIRTUAL PAGE
LRU
STATUS
256
258
260
262
COMBINE
217
PHYSICAL ADDRESS
202
MAIN MEMORY
218

FIG. 7

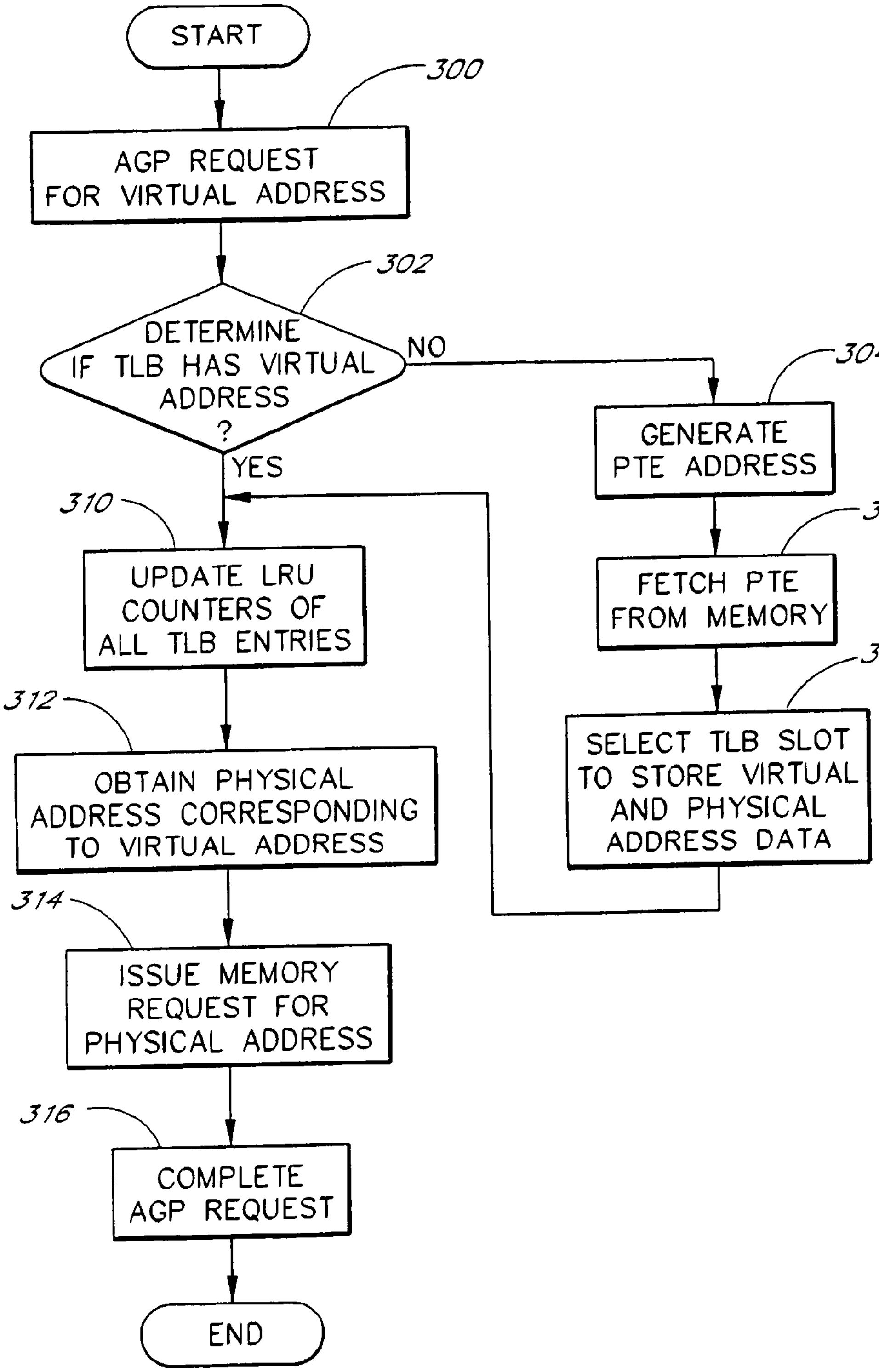

START
300
AGP REQUEST FOR VIRTUAL ADDRESS
302
DETERMINE IF TLB HAS VIRTUAL ADDRESS ?
NO
YES
304
GENERATE PTE ADDRESS
306
FETCH PTE FROM MEMORY
308
SELECT TLB SLOT TO STORE VIRTUAL AND PHYSICAL ADDRESS DATA
310
UPDATE LRU COUNTERS OF ALL TLB ENTRIES
312
OBTAIN PHYSICAL ADDRESS CORRESPONDING TO VIRTUAL ADDRESS
314
ISSUE MEMORY REQUEST FOR PHYSICAL ADDRESS
316
COMPLETE AGP REQUEST
END

FIG. 8

LRU UPDATE PROCESS

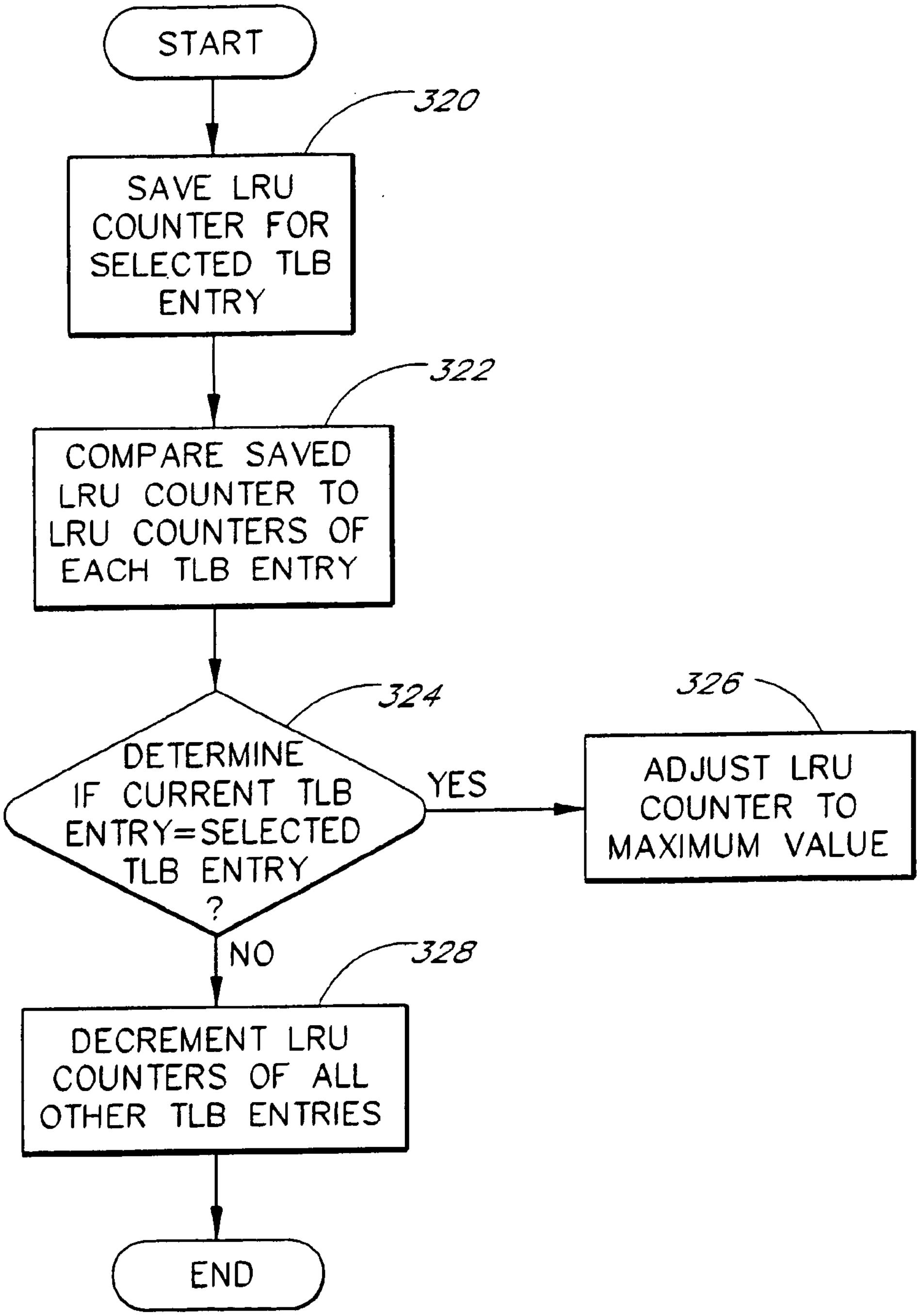

START
320
SAVE LRU COUNTER FOR SELECTED TLB ENTRY
322
COMPARE SAVED LRU COUNTER TO LRU COUNTERS OF EACH TLB ENTRY
324
DETERMINE IF CURRENT TLB ENTRY=SELECTED TLB ENTRY ?
YES
326
ADJUST LRU COUNTER TO MAXIMUM VALUE
NO
328
DECREMENT LRU COUNTERS OF ALL OTHER TLB ENTRIES
END

FIG. 9

TLB SELECTION PROCESS

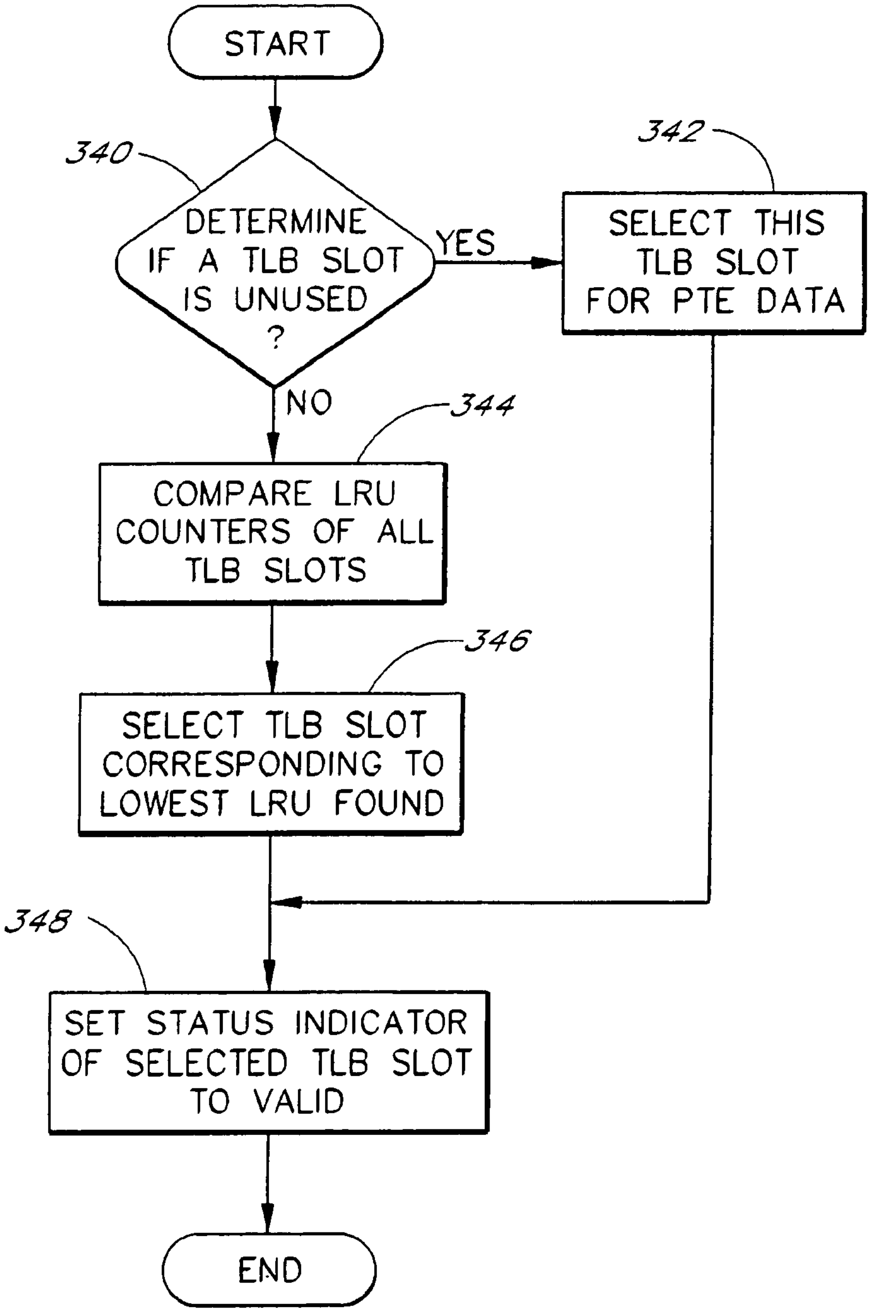

START
340
DETERMINE IF A TLB SLOT IS UNUSED ?
YES
342
SELECT THIS TLB SLOT FOR PTE DATA
NO
344
COMPARE LRU COUNTERS OF ALL TLB SLOTS
346
SELECT TLB SLOT CORRESPONDING TO LOWEST LRU FOUND
348
SET STATUS INDICATOR OF SELECTED TLB SLOT TO VALID
END

FIG. 10

GART PTE FETCH PROCESS

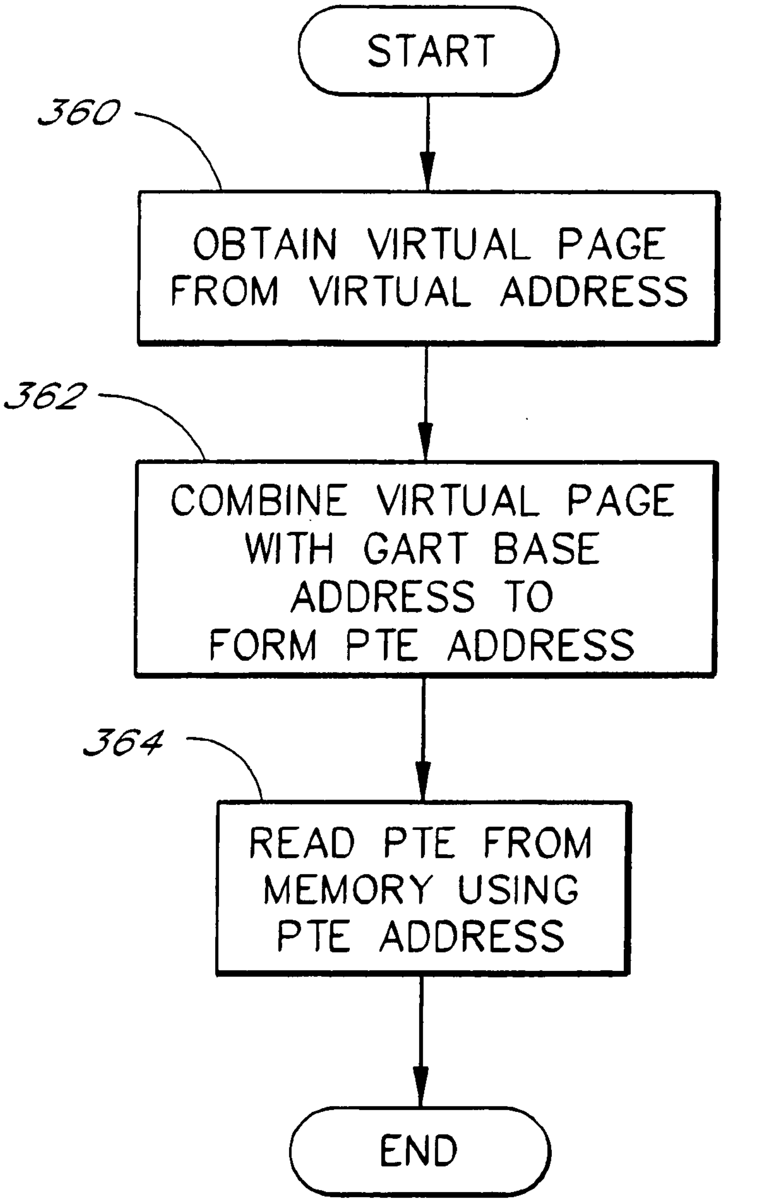

START
360
OBTAIN VIRTUAL PAGE FROM VIRTUAL ADDRESS
362
COMBINE VIRTUAL PAGE WITH GART BASE ADDRESS TO FORM PTE ADDRESS
364
READ PTE FROM MEMORY USING PTE ADDRESS
END

FIG. 11

APPARATUS COMPRISING A TRANSLATION LOOKASIDE BUFFER FOR GRAPHICS ADDRESS REMAPPING OF VIRTUAL ADDRESSES

CROSS REFERENCE TO RELATED APPLICATIONS

This is a division of, and incorporates by reference in its entirety, U.S. application Ser. No. 08/882,054, now U.S. Pat. No. 6,249,853, titled "Apparatus for Graphic Address Remapping", filed Jun. 25, 1997. This application is related to, and incorporates by reference in their entirety, U.S. Pat. No. 6,069,638, filed Jun. 25, 1997, titled "System for Accelerated Graphics Port Address Remapping Interface to Main Memory", U.S. Pat. No. 6,282,625, filed Jun. 25, 1997, titled "Method for Accelerated Graphics Port Address Remapping Interface to Main Memory", U.S. application Ser. No. 09/723,403, filed Nov. 27, 2000, titled "Method for Implementing an Accelerated Graphics Port for a Multiple Memory Controller Computer System", and U.S. Pat. No. 6,252,612, filed Dec. 30, 1997, titled "Accelerated Graphics Port for Multiple Memory Controller Computer Systems".

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to computer systems, and more particularly, to an apparatus for mapping virtual addresses to physical addresses in graphics applications.

2. Description of the Related Technology

As shown in FIG. **1**, a conventional computer system architecture **100** includes a processor **102**, system logic **104**, main memory **106**, a system bus **108**, a graphics accelerator **110** communicating with a local frame buffer **112** and a plurality of peripherals **114**. The processor **102** communicates with main memory **106** through a memory management unit (MMU) in the system logic **104**. Peripherals **114** and the graphics accelerator **110** communicate with main memory **106** and system logic **104** through the system bus **108**. The standard system bus **108** is currently the Peripherals Connection Interface (PCI). The original personal computer bus, the Industry Standard Architecture (ISA), is capable of a peak data transfer rate of 8 megabytes/sec and is still used for low-bandwidth peripherals, such as audio. On the other hand, PCI supports multiple peripheral components and add-in cards at a peak bandwidth of 132 megabytes/sec. Thus, PCI is capable of supporting full motion video playback at 30 frames/sec, true color high-resolution graphics and 100 megabits/sec Ethernet local area networks. However, the emergence of high-bandwidth applications, such as three dimensional (3D) graphics applications, threatens to overload the PCI bus.

For example, a 3D graphics image is formed by taking a two dimensional image and applying, or mapping, it as a surface onto a 3D object. The major kinds of maps include texture maps, which deal with colors and textures, bump maps, which deal with physical surfaces, reflection maps, refraction maps and chrome maps. Moreover, to add realism to a scene, 3D graphics accelerators often employ a z-buffer for hidden line removal and for depth queuing, wherein an intensity value is used to modify the brightness of a pixel as a function of distance. A z-buffer memory can be as large or larger than the memory needed to store two dimensional images. The graphics accelerator **110** retrieves and manipulates image data from the local frame buffer **112**, which is a type of expensive high performance memory. For example, to transfer an average 3D scene (polygon overlap of three) in 16-bit color at 30 frames/sec at 75 Hz screen refresh, estimated bandwidths of 370 megabytes/sec to 840 megabytes/sec are needed for screen resolutions from 640×480 resolution (VGA) to 1024×768 resolution (XGA). Thus, rendering of 3D graphics on a display requires a large amount of bandwidth between the graphics accelerator **110** and the local frame buffer **112**, where 3D texture maps and z-buffer data typically reside.

In addition, many computer systems use virtual memory systems to permit the processor **102** to address more memory than is physically present in the main memory **106**. A virtual memory system allows addressing of very large amounts of memory as though all of that memory were a part of the main memory of the computer system. A virtual memory system allows this even though actual main memory may consist of some substantially lesser amount of storage space than is addressable. For example, main memory may include sixteen megabytes (16,777,216 bytes) of random access memory while a virtual memory addressing system permits the addressing of four gigabytes (4,294,967,296 bytes) of memory.

Virtual memory systems provide this capability using a memory management unit (MMU) to translate virtual memory addresses into their corresponding physical memory addresses, where the desired information actually resides. A particular physical address holding desired information may reside in main memory or in mass storage, such as a tape drive or hard disk. If the physical address of the information is in main memory, the information is readily accessed and utilized. Otherwise, the information referenced by the physical address is in mass storage and the system transfers this information (usually in a block referred to as a page) to main memory for subsequent use. This transfer may require the swapping of other information out of main memory into mass storage in order to make room for the new information. If so, the MMU controls the swapping of information to mass storage.

Pages are the usual mechanism used for addressing information in a virtual memory system. Pages are numbered, and both physical and virtual addresses often include a page number and an offset into the page. Moreover, the physical offset and the virtual offset are typically the same. In order to translate between the virtual and physical addresses, a basic virtual memory system creates a series of lookup tables, called page tables, stored in main memory. These page tables store the virtual address page numbers used by the computer. Stored with each virtual address page number is the corresponding physical address page number which must be accessed to obtain the information. Often, the page tables are so large that they are paged themselves. The page number of any virtual address presented to the memory management unit is compared to the values stored in these tables in order to find a matching virtual address page number for use in retrieving the corresponding physical address page number.

There are often several levels of tables, and the comparison uses a substantial amount of system clock time. For example, to retrieve a physical page address using lookup tables stored in main memory, the typical MMU first looks to a register for the address of a base table which stores pointers to other levels of tables. The MMU retrieves this pointer from the base table and places it in another register. The MMU then uses this pointer to go to the next level of table. This process continues until the physical page address of the information sought is recovered. When the physical address is recovered, it is combined with the offset furnished as a part of the virtual address and the processor uses the result to access the particular information desired. Completion of a typical lookup in the page tables may take from ten to fifteen clock cycles at each level of the search.

To overcome this delay, virtual management systems often include cache memories called translation look aside buffers (TLBs). A TLB is essentially a buffer for caching recently translated virtual page addresses along with their corresponding physical page addresses. Such an address cache works on the same principle as do caches holding data and instructions, the most recently used addresses are more likely to be used than are other addresses. Thus, if a subsequent virtual address refers to the same page as the last one, the page table lookup process is skipped to save time. A TLB entry is like a cache entry wherein a tag portion includes portions of the virtual address and the data portion includes a physical page frame number, protections fields, use bits and status bits. When provided with a virtual page address stored in the TLB (a translation hit), the TLB furnishes a physical page address for the information without having to consult any page lookup tables. When the processor requests a virtual page address not stored in the TLB (a translation miss), the MMU must then consult the page lookup tables. When this occurs, the physical page address recovered is stored along with the virtual page address in the TLB so that it is immediately available for subsequent use. This saves a substantial amount of time on the next use of the information. For example, accessing the information using a TLB may require only one or two clock cycles compared to the hundreds of clock cycles required for a page table lookup.

Virtual memory systems are common in the art. For example, in U.S. Pat. No. 5,446,854, Khalidi et al. disclose a method and apparatus for virtual to physical address translation using hashing. Similarly, Crawford et al. disclose a microprocessor architecture having segmentation mechanisms for translating virtual addresses to physical addresses in U.S. Pat. No. 5,321,836. Lastly, in U.S. Pat. Nos. 5,491,806 and 5,546,555, Horstmann, et al. disclose an optimized translation lookaside buffer for use in a virtual memory system.

As shown in FIG. **1**, moving 3D graphics data to the main memory **106** in current computer systems would require the graphics accelerator **110** to access the 3D graphics data through the PCI system bus **108**. Thus, although Bechtolsheim discloses a data bus enabling virtual memory data transfers in U.S. Pat. Nos. 4,937,734 and 5,121,487, 3D rendering exceeds the peak PCI bandwidth of 132 megabytes/sec because a bandwidth of at least 370 megabytes/sec is needed for data transfer from main memory **106**. Moreover, the graphics accelerator **110** often requires storage of graphics data into large contiguous blocks of memory. For example, a 16-bit 256×256 pixel texture map for 3D graphics applications requires a memory block having a size of 128K bytes. However, operating system software, such as Microsoft®, Windows®, Windows® 95 and Windows NT®, and the system logic **104** often allocate main memory in page frames having smaller sizes, such as 4K. In U.S. Pat. No. 5,465,337, Kong discloses a memory management unit capable of handling virtual address translations for multiple page sizes. However, this does not address the bandwidth limitations of the PCI bus discussed above. In order to move 3D graphics data from the local frame buffer **112** to main memory **106**, computer systems require an improved method for storing and addressing graphics data in main memory.

In U.S. Pat. No. 5,313,577, Meinerth et al. discloses a graphics processor capable of reading from, and writing to, virtual memory. This graphics processor can be described by reference to FIG. **2**, which illustrates a graphics/memory control unit **120** including a graphics processor unit **122** that communicates with a memory control unit **124**. The graphics/memory control unit **120** in turn communicates with the main memory **106** and the frame buffer **112** through a dedicated memory bus **126**. The graphics processor unit **122** includes an address generator and a virtual translation unit to provide for translation of virtual addresses to physical addresses when accessing the main memory **106** and the frame buffer **112**. In addition, the memory control unit **124** communicates with a processor **102** through a dedicated system bus **128**, with an I/O device **114** through a dedicated I/O bus **130** and with computer networks through a dedicated network bus **132**. In contrast to the structure of FIG. **1**, the use of dedicated buses for communication with the main memory **106**, I/O devices **114** and computer networks substantially increases system cost and decreases the flexibility with which a computer system can be upgraded. For example, to upgrade the graphics capability of a computer system having the structure as illustrated in FIG. **1**, one simply connects a more powerful graphics adapter to the PCI bus **108** (FIG. **1**). However, upgrading the graphics capability of a computer system having the structure of FIG. **2** requires replacement of the memory control unit **124** as well as the graphics processor unit **122**. Similarly, the structure of FIG. **2** is not compatible with the vast majority of available PCI enhancement devices. Moreover, the structure of FIG. **2** also requires the graphics processor unit **122** to access 3D graphics data through a memory bus **126**.

In view of the limitations discussed above, computer manufacturers require a modular architecture that reduces the cost of system upgrades, such as enhanced 3D graphics adapters, to improve display performance. Similarly, to reduce system memory costs, computer manufacturers require improved methods for storing, addressing and retrieving graphics data from main memory instead of expensive local frame buffer memory. Moreover, to address the needs of high bandwidth graphics applications without substantial increases in system cost, computer manufacturers require improved technology to overcome current system bus bandwidth limitations.

SUMMARY OF THE INVENTION

One embodiment of the invention includes a graphics address remapping table (GART), the GART stored in memory, comprising at least one page table entry (PTE) providing information for translation of a virtual address to a physical address, wherein the virtual address includes a first portion and a second portion, the first portion being used to locate a PTE in the GART corresponding to the virtual address and wherein the second portion and the information provided by the PTE are combined to provide the physical address.

Another embodiment of the invention includes a page table entry for a graphics address remapping table stored in memory comprising a physical page translation field having translation information and a feature bits field having at least one indicator defining an attribute of the physical page translation field.

Yet another embodiment of the invention includes a translation lookaside buffer (TLB) in a memory, the TLB receiving a portion of a virtual address selected from a graphics address remapping range, comprising at least one TLB entry, wherein each of the at least one TLB entries includes a virtual page field and a corresponding physical page field, wherein if the portion of the virtual address matches the virtual page field of one TLB entry, the TLB provides translation information from the physical page field of the one TLB entry to form a physical address.

Yet another embodiment of the invention includes an apparatus for graphic address remapping of a virtual address comprising a graphics address remapping table (GART) stored in memory and having information which is used to translate the virtual address to a physical address and a translation lookaside buffer (TLB) receiving a portion of the virtual address, the TLB having at least one TLB entry, wherein each of the at least one TLB entries includes a virtual page field and a corresponding physical page field, wherein if the portion of the virtual address matches the virtual page field of one TLB entry, the TLB provides translation information from the physical page field of the one TLB entry to form the physical address and wherein if the portion of the virtual address does not match the virtual page field of one TLB entry, the GART provides translation information referenced by the portion of the virtual address to form the physical address.

Lastly, yet another embodiment of the present invention includes an apparatus for graphic address remapping of a virtual address comprising an interface and a translation lookaside buffer (TLB) in communication with the interface, the TLB having at least one TLB entry including information which is used to translate the virtual address to a physical address, wherein the interface receives a portion of the virtual address and provides access to a TLB entry corresponding to the portion of the virtual address.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram illustrating the architecture of a prior art computer system.

FIG. **2** is a block diagram illustrating the architecture of another prior art computer system.

FIG. **3** is a block diagram illustrating the architecture of a computer system of one embodiment of the present invention.

FIG. **4** is a diagram illustrating the address space of a processor of one embodiment of the present invention.

FIG. **5***a* is a diagram illustrating the translation of a virtual address to a physical address of one embodiment of the present invention.

FIG. **5***b* is a diagram illustrating a page table entry (PTE) of the graphic address remapping table (GART) of one embodiment of the present invention.

FIG. **6***a* is a diagram illustrating the generation of a translation look aside buffer (TLB) entry of one embodiment of the present invention.

FIG. **6***b* is a block diagram illustrating one embodiment of an interface for the direct access of a translation look aside buffer (TLB) of one embodiment of the present invention.

FIG. **7** is a diagram illustrating the translation of a virtual address to a physical address using the TLB of one embodiment of the present invention.

FIG. **8** is a flowchart illustrating the method of processing an AGP request of the present invention.

FIG. **9** is a flowchart illustrating the method of updating a least recently used (LRU) counter of one embodiment of the present invention.

FIG. **10** is a flowchart illustrating the method of selecting a slot to store a TLB entry of one embodiment of the present invention.

FIG. **11** is a flowchart illustrating the method of fetching a page table entry (PTE) of one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

In contrast to the conventional computer system architecture **100** (FIG. **1**), embodiments of the present invention enables relocation of a portion of the 3D graphics data, such as the texture data, from the local frame buffer **112** (FIG. 1) to main memory **106** (FIG. 1) to reduce the size, and thus the cost, of the local frame buffer **112** and to improve system performance. For example, as texture data is generally read only, moving it to main memory **106** does not cause coherency or data consistency problems. Similarly, as texture data is loaded from mass storage into main memory **106**, leaving it in main memory **106** instead of copying it to the local frame buffer **112** reduces overhead. Moreover, as the complexity and quality of 3D images has increased, leaving 3D graphics data in the local frame buffer **112** has served to increase the computer system cost over time. Thus, although moving 3D graphics data to main memory **106** may likewise require an increase in the size of the main memory **106**, the architecture of the present invention reduces the total system cost because it is less expensive to increase main memory **106** than to increase local frame buffer memory **112**.

Referring now to FIG. **3**, the computer system architecture **150** of one embodiment of the present invention includes a processor **152**, system logic **154**, main memory **156**, a system bus **158**, a graphics accelerator **160** communicating with a local frame buffer **162** and a plurality of peripherals **164**. The processor **152** communicates with the main memory **156** through a memory management unit (MMU) in the system logic **154**. Peripherals **114** communicate with the main memory **156** and system logic **154** through the system bus **158**. Note however that the graphics accelerator **160** communicates with the system logic **154** and main memory **156** through an accelerated graphics port (AGP) **166**. AGP **166** is not a bus, but a point-to-point connection between an AGP compliant target, the MMU portion of the system logic **154**, and an AGP-compliant master, the graphics accelerator **160**. The AGP **166** point-to-point connection enables data transfer on both the rising and falling clock edges, improves data integrity, simplifies AGP protocols and eliminates bus arbitration overhead. AGP provides a protocol enhancement enabling pipelining for read and write accesses to main memory **156**.

For example, the graphics accelerator **160** initiates a pipelined transaction with an access request. System logic **154** responds to the request by initiating the corresponding data transfer at a later time. The graphics accelerator **160** can then issue its next pipelined transaction while waiting for the previous data to return. This overlap results in several read or write requests being active at any point in time. In one embodiment, the AGP **166** operates with a 66 MHz clock using 3.3 volt signaling. As data transfers can occur on both clock edges, the AGP **166** enables an effective 133 MHz data transfer rate and can reach a peak bandwidth of 533 megabytes/sec. For detailed information on the AGP **166**, consult the Accelerated Graphics Port Interface Specification, Revision 1.0, released by Intel Corporation and available from Intel in Adobe¤ Acrobat¤ format on the World Wide Web . This document is hereby incorporated by reference.

As noted above, the embodiment of FIG. **3** enables the graphics accelerator **160** to access both main memory **156** and the local frame buffer **162**. From the perspective of the graphics accelerator **160**, the main memory **156** and the local frame buffer **162** are logically equivalent. Thus, to optimize system performance, graphics data may be stored in either the main memory **156** or the local frame buffer **162**. In contrast to the direct memory access (DMA) model where graphics data is copied from the main memory **156** into the local frame buffer **162** by a long sequential block transfer prior to use, the graphics accelerator **160** of the present invention can also use, or "execute," graphics data directly from the memory in which it resides (the "execute" model). However, since the main memory **156** is dynamically allocated in random pages of a selected size, such as 4K, the "execute" model requires an address mapping mechanism to map random pages into a single contiguous, physical address space needed by the graphics accelerator **160**.

FIG. **4** illustrates an embodiment of the address space **180** of the computer system **150** (FIG. **3**) of the present invention. For example, a 32 bit processor **152** (FIG. **3**) has an address space **180** including 2^{32} (or 4,294,967,296) different addresses. A computer system **150** (FIG. **3**) typically uses different ranges of the address space **180** for different devices and system agents. In one embodiment, the address space **180** includes a local frame buffer range **182**, a graphics address remapping table (GART) range **184** and a main memory range **186**. In contrast to prior art systems, addresses falling within the GART range **184** are remapped to non-contiguous pages within the main memory range **186**. All addresses not in the GART range **184** are passed through without modification so that they map directly to the main memory range **186** or to device specific ranges, such as the local frame buffer range **182**. In one embodiment, the system logic **154** performs the address remapping using a memory based table, the GART, defined in software with an application program interface (API). Moreover, the GART table format is abstracted to the API by a hardware abstraction layer (HAL) or a miniport driver provided by the system logic **154**. Thus, by defining the GART in software, the present invention advantageously provides the substantial implementation flexibility needed to address future partitioning and remapping circuitry (hardware) as well as any current or future compatibility issues.

FIG. **5***a* illustrates the translation of a virtual address **200** to a physical address **202** in one embodiment of the present invention. As discussed previously, in one embodiment, only those virtual addresses falling within the GART range **184** (FIG. **4**) are remapped to main memory **186** (FIG. **4**). A virtual address **200** includes a virtual page number field **204** and an offset field **206**. Translation of the contents of the virtual page number field **204** occurs by finding a page table entry (PTE) corresponding to the virtual page number field **204** among the plurality of GART PTEs **208** in the GART table **210**. To identify the appropriate PTE having the physical address translation, the GART base address **212** is combined at **213** with the contents of the virtual page number field **204** to obtain a PTE address **214**. The contents referenced by the PTE address **214** provide the physical page number **216** corresponding to the virtual page number **204**. The physical page number **216** is then combined at **217** with the contents of the offset field **206** to form the physical address **202**. The physical address **202** in turn references a location in main memory **218** having the desired information.

The GART table **210** may include a plurality of PTEs **208** having a size corresponding to the memory page size used by the processor **152** (FIG. **3**). For example, an Intel¤ Pentium¤ or Pentium¤ Pro processor operates on memory pages having a size of 4K. Thus, a GART table **210** adapted for use with these processors may include PTEs referencing 4K pages. In one embodiment, the virtual page number field **204** comprises the upper 20 bits and the offset field **206** comprises the lower 12 bits of a 32 bit virtual address **200**. Thus, each page includes 2^{12}=4096 (4K) addresses and the lower 12 bits of the offset field **206** locate the desired information within a page referenced by the upper 20 bits of the virtual page number field **204**. The GART table **210** preferably resides in the main memory **218**. Memory refers generally to storage devices, such as registers, SRAM, DRAM, flash memory, magnetic storage devices, optical storage devices and other forms of volatile and non-volatile storage.

FIG. **5***b* illustrates one possible format for a GART PTE **220**. The GART PTE **220** includes a feature bits field **222** and a physical page translation (PPT) field **224**. In contrast to prior art systems where hardwired circuitry defines a page table format, the GART table **210** (FIG. **5***a*) may include PTEs of configurable length enabling optimization of table size and the use of feature bits defined by software. The length of the GART PTE **220** is $2^{PTESize}$ bytes or $8*2^{PTESize}$ bits. For example, for a PTESize=5, the GART PTE has a length of 32 bytes or 256 bits. The PPT field **224** includes PPTSize bits to generate a physical address **202** (FIG. **5***a*). PPTSize defines the number of translatable addresses, and hence the GART table **210** (FIG. **5***a*) includes $2^{PPTSize}$ PTE entries. As PTESize defines the size of each GART PTE **220**, the memory space needed for the entire GART table **210** (FIG. **5***a*) is $2^{(PTESize+PPTSize)}$ bytes. For example, the GART table **210** in a system with a 4K (=2^{12}) memory page size and 32 megabytes (=2^{25}) of main memory **218** (FIG. **5***a*) includes $2^{25}/2^{12}=2^{13}$=8192 PTEs. Thus, only 13 bits are needed to define 8192 unique PTEs to span the entire 32 megabytes of main memory **218** (FIG. **5***a*) and PPTSize=13. However, to accommodate various software feature bits, each PTE may have a size of 8 bytes (=2^3 and PTESize=3). Thus, the size of the GART table **210** is $2^{(PTESize+PPTSize)}=2^{(3+13)}=2^{16}$=65536 bytes=64K.

As noted above, the GART table **210** (FIG. **5***a*) may use 4K page boundaries. Thus, when (PTESize+PPTSize) is less than 12 bits (2^{12}=4096 bytes=4K), the entire GART table **210** (FIG. **5***a*) resides within one 4K page. For values greater than 12, the GART table **210** (FIG. **5***a*) resides on multiple 4K pages. To maintain compatibility with the Intel¤ Pentium¤ Pro processor caches, the GART base address **214** (FIG. **5***a*) may begin on a $2^{(PTESize+PPTSize)}$ byte boundary. Thus, a GART base address **214** (FIG. **5***a*) can not have a value which aligns the GART table **210** (FIG. **5***a*) on an address boundary less than the size of the GART table **210** (FIG. **5***a*). For example, an 8K GART table **210** (FIG. **5***a*) must begin on a 8K boundary.

In one embodiment, an initialization BIOS implements the GART table **210** (FIG. **5***a*) by loading configuration registers in the system logic **154** (FIG. **3**) during system boot up. In another embodiment, the operating system implements the GART table **210** (FIG. **5***a*) using an API to load the configuration registers in the system logic **154** (FIG. **3**) during system boot up. The operating system then determines the physical location of the GART table **210** (FIG. **5***a*) within main memory **218** (FIG. **5***a*) by selecting the proper page boundary as described above (i.e., an 8K GART table begins on an 8K boundary). For example, the system loads configuration registers holding the GART base address **214** (FIG. **5***a*) defining the beginning of the GART table **210** (FIG. **5***a*), PTESize defining the size of a GART PTE **220** and PPTSize defining the size of the physical address used to translate a virtual address. In addition, the system loads a configuration register forAGPAperture, defining the lowest address of the GART range **184** (FIG. **4**), and PhysBase, defining the remaining bits needed to translate a virtual address not included in the PPTSize bits.

For example, consider a system having 64 megabytes of main memory **218** (FIG. **5***a*) encompassing physical addresses 0 through 0x03FFFFFF with the AGP related data occupying the upper 32 megabytes of main memory **218** referenced by physical addresses 0x02000000 through 0x03FFFFFF. If the GART Range **184** (FIG. **4**) begins at the 256 megabyte virtual address boundary 0x10000000, the invention enables translation of virtual addresses within the GART Range **184** to physical addresses in the upper 32 megabytes of main memory **218** corresponding to physical addresses in the range 0x02000000 through 0x03FFFFFF. As noted earlier, a GART table **210** includes multiple PTEs, each having physical page translation information **224** and software feature bits **222**. The GART table **210** may be located at any physical address in the main memory **218**, such as the 2 megabyte physical address 0x00200000. For a system having a 4K memory page size and a GART PTE **220** of 8 byte length, the GART table **210** is configured as follows:

PhysBase	:=0x02000000	—Start of remapped physical address
PhysSize	:=32 megabytes	—Size of remapped physical addresses
AGPAperture	:=0x10000000	—Start address of GART Range
GARTBase	:=0x00200000	—Start address of GART table
$2^{PTESize}$	:=8 bytes	—Size of each GART Page Table Entry
PageSize	:=4 kilobytes	—Memory page size

To determine the number of PTEs in the GART table **210**, the size of the physical address space in main memory **218** allocated to AGP related data, the upper 32 megabytes=33554432 bytes, is divided by the memory page size, 4K=4096 bytes, to obtain 8192 PTEs. Note that $8192=2^{13}=2^{PTESize}$ and thus, PTESize=13. To implement the GART table **210**, the configuration registers are programmed with the following values:

PhysBase	:=0x02000000	—Start of remapped physical address
AGPAperture	:=0x10000000	—Start address of GART Range
GARTBase	:=0x00200000	—Start address of GART table
PTESize	:=3	—Size of each GART PTE
PPTSize	:=13	—Number of PPT bits in each PTE

Lastly, the GART table **210** is initialized for subsequent use.

Using pseudo-VHDL code, system logic **154** (FIG. **3**) can quickly determine whether a 32 bit AGP address (AGPAddr) requires translation from a virtual to physical address (PhysAddr) as follows:

```
if ((AGPAddr(31 downto 12) and not ($2^{PPTSize\ -1)}$)=
    AGPAperture (31 downto 12)) then
    Virtual=true;
else
    Virtual=false;
end if;
```

When the address is virtual, then the PTE address 214 (PTEAddr) is calculated as follows:

```
PTEAddr<=((AGPAddr(31 downto 12) and ($2^{(PPTSize)}$-
    1)) shl $2^{PTESize}$) or (GARTBase and not ($2^{(PTESize+PPTSize)}$-1)));
```

Note that the "shl" function indicates a left shift with zero fill, which can be implemented in hardware using a multiplexer. Lastly, to determine the physical address **202** (PhysAddr) when PPTSize does not include sufficient bits to remap the entire GART range **184** (FIG. **4**), the physical page **216** is generated as follows:

```
PhysAddr(31 downto 12)<=(PhysBase(31 downto 12)
    and not ($2^{PPTSize}$-1)) or (PTE and ($2^{PPTSize}$-1)));
```

To obtain the physical address **202**, the physical page **216**, PhysAddr(31 downto 12), is then combined with the offset **206**. Note that the pseudo-code above avoids the use of adders, which impede system performance at high clock frequencies, in the virtual to physical address translation process.

To illustrate the use of the pseudo-code above, suppose an AGP master, such as the graphics accelerator **160** (FIG. **3**), presents the virtual address 0x0002030, which corresponds to AGPAddr in the pseudo-code, to the system logic **154** (FIG. **3**) for translation. To determine if AGPAddr=0x10002030 is appropriate for translation using the GART table configured above, the system logic **154** first evaluates the if condition:

```
((AGPAddr(31 downto 12) and not ($2^{PPTSize}$-1))=
    AGPAperture (31 downto 12))
```

to determine if it is true or false. In addition, the expression ($2^{PPTSize}$-1) indicates that the lower PPTSize bits are set, which is easily performed in hardware. For the GART table **210** configured above, note that PPTSize=13, ($2^{PPTSize}$-1)=0x01IFFF (hexadecimal) and AGPAperture=0x10000000. The notation (31 downto 12) indicates use of bit positions 12 through 31 of an address, which is equivalent to truncating the lower 12 bits of a binary address or the lower three values of a hexadecimal address. Thus, for AGPAddr=0x10002030 and AGPAperture=0x10000000, AGPAddr(31 downto 12)=0x10002 and AGPAperture(31 downto 12)=0x10000. Now, substitute the values for AGPAddr, AGPAperture and ($2^{PPTSize}$-1) into the if condition:

```
((AGPAddr(31 downto 12) and not ($2^{PPTSize}$-1))=
    AGPAperture (31 downto 12)) -or-
(0x10002 and not (0x01FFF))=0x10000 -or-
0x10000=0x10000
```

to calculate a result. Here, the result is true indicating that AGPAddr=0x10002030 is a valid address for translation. Similarly, for the virtual address 0x11002030, the if condition produces this result: 0x11000=0x10000. As 0x110000x10000, this result is false indicating that the virtual address 0x11002030 does not fall within the GART range **184**. If an AGP master presented the virtual address 0x11002030, the system logic **154** reports an error.

To determine the location of the PTE in the GART table **210** having the translation information for the virtual address AGPAddr=0x10002030, the expression:

```
PTEAddr<=((AGPAddr(31 downto 12) and ($2^{(PPTSize)}$-
    1)) shl $2^{PTESize}$) or (GARTBase and not ($2^{(PTESize+PPTSize)}$-1)))
```

is evaluated. For the GART table **210** configured above, GARTBase=0x00200000, PPTSize=13, PTESize=3 and ($2^{(PTESize+PPTSize)}$-1)=0x0FFFF. As noted above, ($2^{PPTSize}$-1)=0x01FFF and AGPAddr(31 downto 12)=0x10002. Now, substitute the values into the equation for PTEAddress:

```
PTEAddr<=((0x10002 and 0x01FFF) shl 3) or
    (0x00200000 and not (0x0FFFF)) -or-
PTEAddr<=(0x00002 shl 3) or (0x00200000) -or-
PTEAddr<=(0x00000010) or (0x00200000)=
    0x00200010.
```

As each PTE occupies 8 bytes and the GART table **210** begins at the GARTBase address=0x00200000, the calculated PTEAddress=0x00200010 corresponds to the third entry or PTE(2), 16 bytes away from the GARTBase address. Suppose that the lower 32 bits (or 4 bytes) of the value at PTE(2)=0x12345678. As shown in the embodiment of FIG. 5*b*, the lower PPTSize=13 bits correspond to the PPT translation bits and the higher order bits are software feature bits **222**. Of course, in another embodiment, the PPT translation information may comprise the higher order bits while the software feature bits **222** may comprise the lower order bits. Moreover, the PPT translation information and the software feature bits **222** may be located at any of the bit positions within a PTE **220**.

Lastly, to calculate the physical address corresponding to the virtual address AGPAddr=0x10002030, the expression:

PhysAddr(31 downto 12)<=(PhysBase(31 downto 12) and not ($2^{PPTSize}$−1)) or (PTE and ($2^{PPTSize}$−1)))

is evaluated. For the GART table **210** configured above, PhysBase=0x02000000 and ($2^{PPTSize}$−1)=0x01FFF. Note also that PTE(2)=0x12345678. Now, substitute the values into the equation for PhysAddr(31 downto 12):

PhysAddr(31 downto 12)<=(0x02000 and not (0x01FFF)) or (0x12345678 and 0x01FFF)) -or-

PhysAddr(31 downto 12)<=(0x02000) or (0x00001678)= 0x03678. Note that the offset 206 corresponds to the lower 12 bits of the virtual address 0x10002030 or AGPAddr(11 downto 0)=030. Thus, to obtain the physical address **206**, the physical page **216** is combined with the offset **206** to form PhysAddr(31 downto 0) or 0x03678030. To summarize, the pseudo-code of the embodiment described above illustrates the translation of the virtual address 0x10002030 to the physical address 0x03678030.

Moreover, the feature bits field **222** provides status information for use in virtual to physical address translations. In contrast to prior art systems, the feature bits of one embodiment of the present invention provide substantial design flexibility by enabling software to change the format of the GART table **210** (FIG. 5*a*) without the need for a costly redesign of the hardwired circuitry. For example, during an address translation, the system may need to verify that the physical address corresponding to the virtual address still includes valid data. Similarly, the system may need to determine if a referenced physical address has been read or written to. The contents of the feature bits field **222** provide this functionality. In one embodiment, the feature bits field **222** includes indicators for PTE valid **226**, page read **228** and page write **230**. These indicators **226**, **228**, **230** may be located anywhere within the feature bits field **222** and may be implemented using at least one bit. To implement an indicator, such as PTE valid **226**, the present invention uses a mask register loaded during system boot up. Thus, for PTE valid **226**, the ValidMask register is used to select the bit(s) to set in the feature bits field **222** to indicate a valid PTE. Similarly, for page read **228**, the ReadMask register is used to select the bit(s) to set when a translated address has been read. Furthermore, for a page write **230**, the WriteMask register is used to select the bit(s) to set when a translated address has been written to. For example, if ValidMask is zero, then no PTE Valid **226** indicator is defined. Otherwise, a PTE Valid **226** mask is defined and can be applied to a GART PTE **220** to determine if the physical address translation is valid. The following VHDL pseudo-code implements this logic:

```
if ((ValidMask=0) or ((ValidMask and PTE)=ValidMask))
  then
  PTEValid :=true;
else
  PTEValid :=false;
end if;
```

Similarly, to implement the page read **228** and page write **230** indicators, a logical OR operation is performed on the GART PTE **220** using the WriteMask during write operations and with the ReadMask during read operations. The resulting GART PTE **220** is then written to memory **218** (FIG. 5*a*) to provide the page read **228** or page write **230** status information. In a similar fashion, if the WriteMask or ReadMask is zero, then no page write **230** or page read **228** indicator is defined and the GART PTE **220** is not written to memory. The following VHDL pseudo-code implements the page write **230** and page read **228** indicators:

```
if ((WriteMask 0) and ((PTE and WriteMask) WriteMask))
then
  PTE :=PTE or WriteMask;
  UpdatePTE :=true;
else
  PTE :=PTE;
  UpdatePTE :=false;
end if;
if ((ReadMask 0) and ((PTE and ReadMask) ReadMask))
then
  PTE :=PTE or ReadMask;
  UpdatePTE :=true;
else
  PTE :=PTE;
  UpdatePTE :=false;
end if;
```

As discussed previously, the indicators **226**, **228**, **230** may be implemented by programming a mask register during system boot up. In one embodiment, the initialization BIOS programs the mask register. In another embodiment, an operating system API programs the mask register during system boot up.

For example, suppose the following mask registers:

ValidMask	:=0x00100000	—Position of Valid indicator in PTE
WriteMask	:=0x00200000	—Position of Write indicator in PTE
ReadMask	:=0x00400000	—Position of Read indicator in PTE

are programmed during system boot up. To determine if the contents of a PTE **220** are valid, the if condition:

((ValidMask=0) or ((ValidMask and PTE)=ValidMask)) is evaluated to determine if it is true or false. Referring back to the previous example, note that PTE(2)= 0x12345678. Now, substitute the values of PTE(2) and ValidMask into the if condition:

((ValidMask=0) or ((ValidMask and PTE)=ValidMask)) -or-

((0x00100000=0) or ((0x00100000 and 0x12345678)= 0x00100000)) -or-

((0x00100000=0) or (0x00100000=0x00100000))

to calculate a result. Here, the result is true indicating that the PTE is valid. Similarly, for a ValidMask set to 0x01000000, evaluation of the if condition proceeds as follows:

((ValidMask=0) or ((ValidMask and PTE)=ValidMask)) -or-

((0x01000000=0) or ((0x01000000 and 0x12345678)= 0x01000000)) -or-

((0x0100000=0) or (0x00000000=0x01000000))

to produce a false result as both (0x0100000 0) and (0x00000000 0x01000000), indicating an error reporting

US006418523B2

(12) United States Patent
Porterfield

(10) Patent No.: US 6,418,523 B2
(45) Date of Patent: Jul. 9, 2002

(54) APPARATUS COMPRISING A TRANSLATION LOOKASIDE BUFFER FOR GRAPHICS ADDRESS REMAPPING OF VIRTUAL ADDRESSES

(75) Inventor: **A. Kent Porterfield**, New Brighton, MN (US)

(73) Assignee: **Micron Electronics, Inc.**, Nampa, ID (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/865,653**

(22) Filed: **May 24, 2001**

Related U.S. Application Data

(62) Division of application No. 08/882,054, filed on Jun. 25, 1997, now Pat. No. 6,249,853.

(51) **Int. Cl.**^{7} **G06F 12/10**
(52) **U.S. Cl.** **711/207**; 345/568
(58) **Field of Search** 711/202, 203, 711/206, 207, 208; 345/568

(56) References Cited

U.S. PATENT DOCUMENTS

4,016,545	A	4/1977	Lipovski
4,507,730	A	3/1985	Johnson et al.
4,937,734	A	6/1990	Bechtolsheim
4,969,122	A	11/1990	Jensen
5,121,487	A	6/1992	Bechtolsheim
5,133,058	A	7/1992	Jensen
5,155,816	A	10/1992	Kohn
5,222,222	A	6/1993	Mehring et al.
5,263,142	A	11/1993	Watkins et al.
5,265,213	A	11/1993	Weiser et al.
5,265,227	A	11/1993	Kohn et al.
5,265,236	A	11/1993	Mehring et al.
5,305,444	A	4/1994	Becker et al.
5,313,577	A	5/1994	Meinerth et al.
5,315,696	A	5/1994	Case et al.
5,315,698	A	5/1994	Case et al.
5,321,806	A	6/1994	Meinerth et al.
5,321,807	A	6/1994	Mumford
5,321,836	A	6/1994	Crawford et al.
5,361,340	A	11/1994	Kelly et al.
5,392,393	A	2/1995	Deering
5,396,614	A	3/1995	Khalidi et al.
5,408,605	A	4/1995	Deering

(List continued on next page.)

OTHER PUBLICATIONS

Accelerated Graphics Port Interface Specification. Revision 1.0 Intel Corporation. Jul. 31, 1996. 81 pgs.

Intel Advance information "INTEL 440LX AGPSET:82443LX PCI A.G.P. Controller (PAC)" Aug. 97, 139 pp.

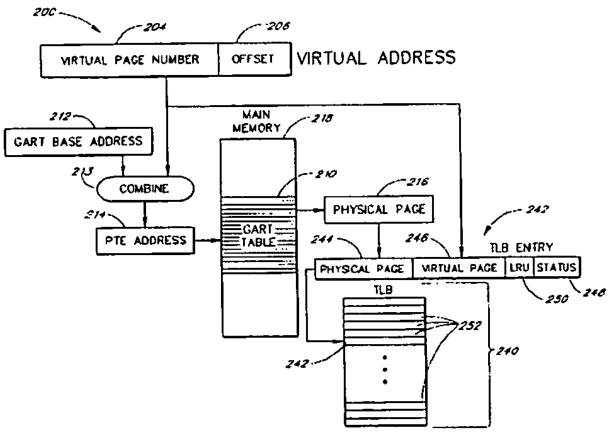
LSI Logic L64852 Mbus–to–Sbus Controller (M2S) Technical Manual. LSI Logic Corporation (1993). 73 pp.

Primary Examiner—Kevin Verbrugge
(74) *Attorney, Agent, or Firm*—Knobbe, Martens, Olson & Bear LLP

(57) ABSTRACT

A modular architecture for storing, addressing and retrieving graphics data from main memory instead of expensive local frame buffer memory. A graphic address remapping table (GART), defined in software, is used to remap virtual addresses falling within a selected range, the GART range, to non-contiguous pages in main memory. Virtual address not within the selected range are passed without modification. The GART includes page table entries (PTEs) having translation information to remap virtual addresses falling within the GART range to their corresponding physical addresses. The GART PTEs are of configurable length enabling optimization of GART size and the use of feature bits, such as status indicators, defined by software. The GART is implemented during system boot up by configuration registers. Similarly, the PTEs are configured using mask registers. The GART may be used in conjunction with a translation lookaside buffer (TLB) to improve address remapping performance.

18 Claims, 13 Drawing Sheets



U.S. PATENT DOCUMENTS

5,426,750	A	6/1995	Becker et al.
5,440,682	A	8/1995	Deering
5,446,854	A	8/1995	Khalidi et al.
5,465,337	A	11/1995	Kong
5,479,627	A	12/1995	Khalidi et al.
5,491,806	A	2/1996	Horstmann et al.
5,500,948	A	3/1996	Hinton et al.
5,524,233	A	6/1996	Milburn et al.
5,542,062	A	7/1996	Taylor et al.
5,546,555	A	8/1996	Horstmann et al.
5,548,739	A	8/1996	Yung
5,553,023	A	9/1996	Lau et al.
5,584,014	A	12/1996	Nayfeh et al.
5,586,283	A	12/1996	Lopez-Aguado et al.
5,664,161	A	9/1997	Fukushima et al.
5,675,750	A	10/1997	Matsumoto et al.
5,737,765	A	4/1998	Shigeeda
5,778,197	A	7/1998	Dunham
5,815,167	A	9/1998	Muthal et al.
5,845,327	A	12/1998	Rickard et al.
5,854,637	A	12/1998	Sturgess
5,861,893	A	1/1999	Sturgess
5,889,970	A	3/1999	Horan et al.
5,892,964	A	4/1999	Horan et al.
5,909,559	A	6/1999	So
5,911,051	A	6/1999	Carson et al.

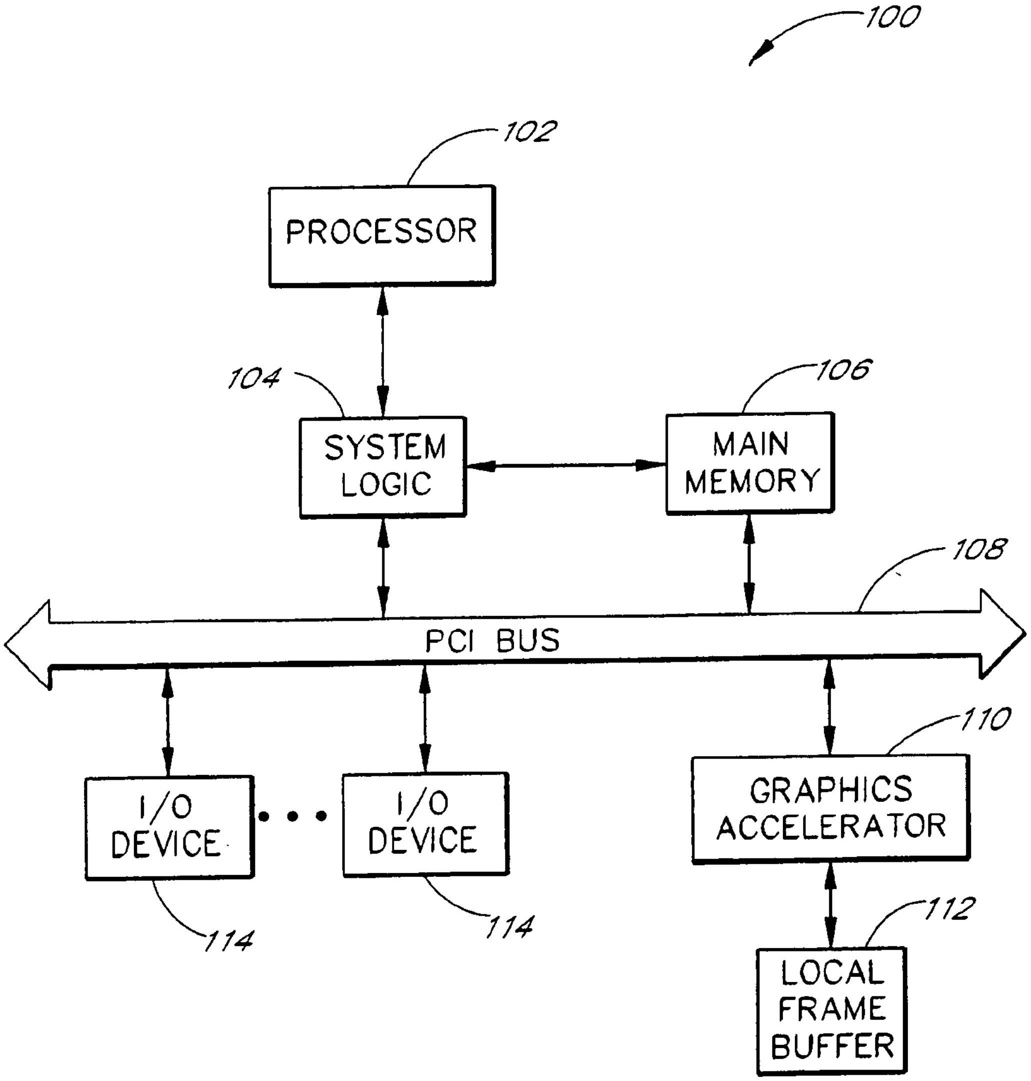


FIG. 1
(PRIOR ART)

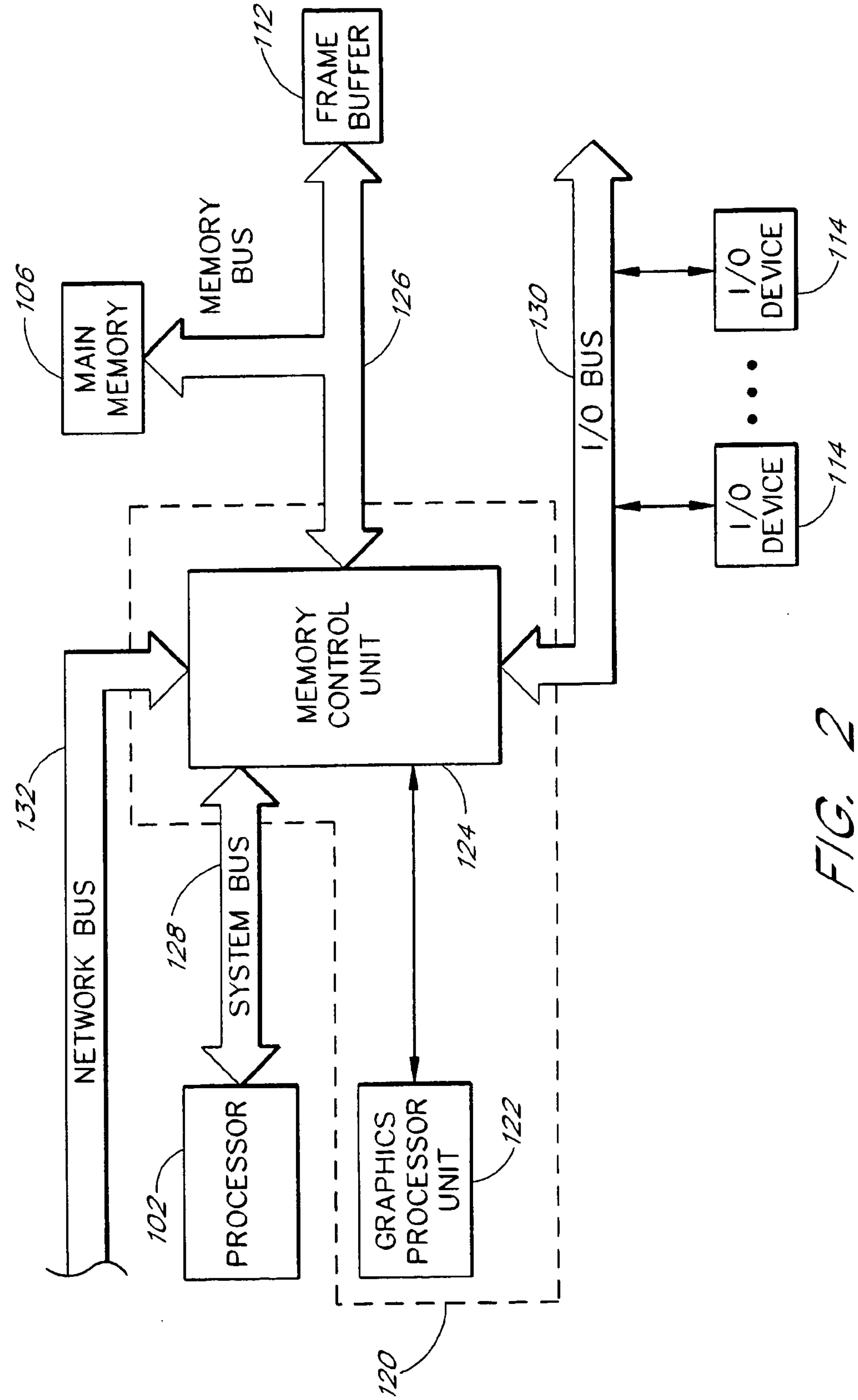


FIG. 2
(PRIOR ART)

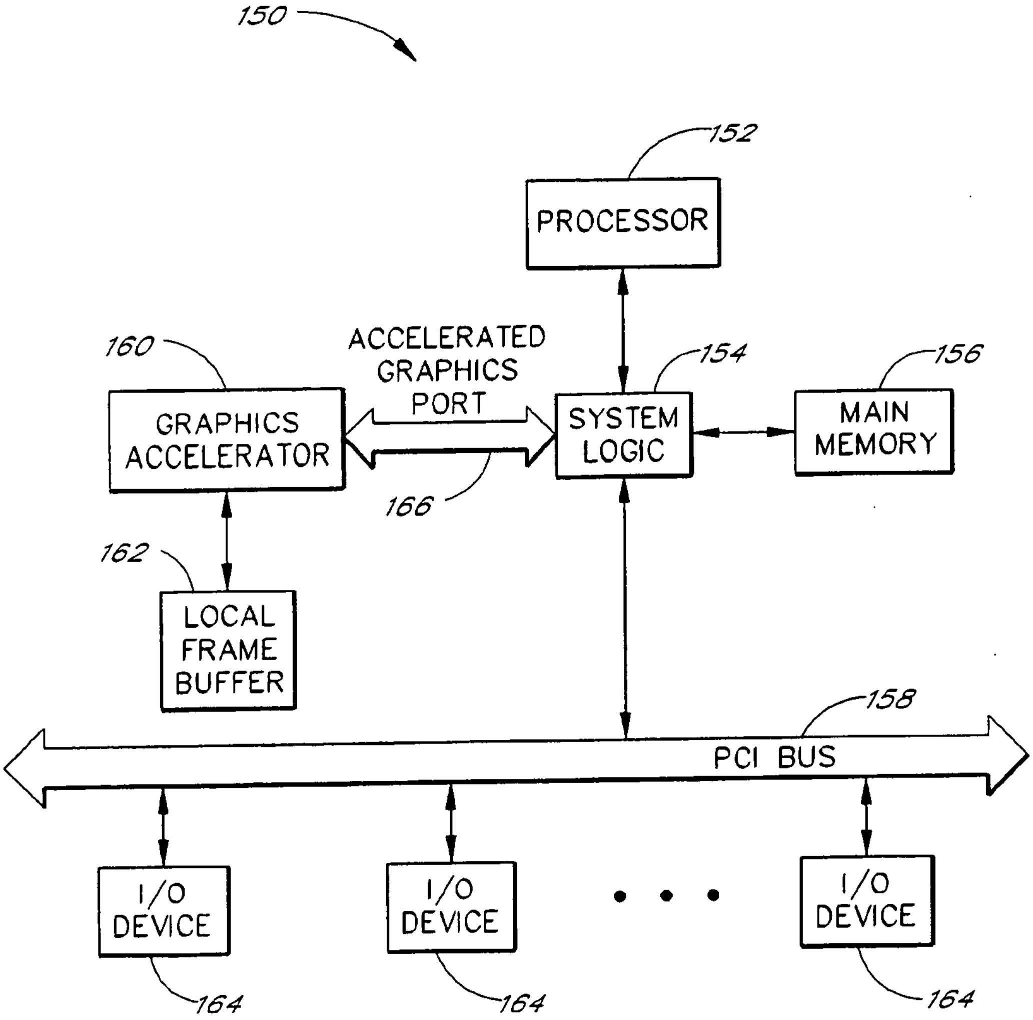

150
152
PROCESSOR
160
GRAPHICS ACCELERATOR
ACCELERATED GRAPHICS PORT
166
154
SYSTEM LOGIC
156
MAIN MEMORY
162
LOCAL FRAME BUFFER
158
PCI BUS
I/O DEVICE
I/O DEVICE
• • •
I/O DEVICE
164
164
164

FIG. 3

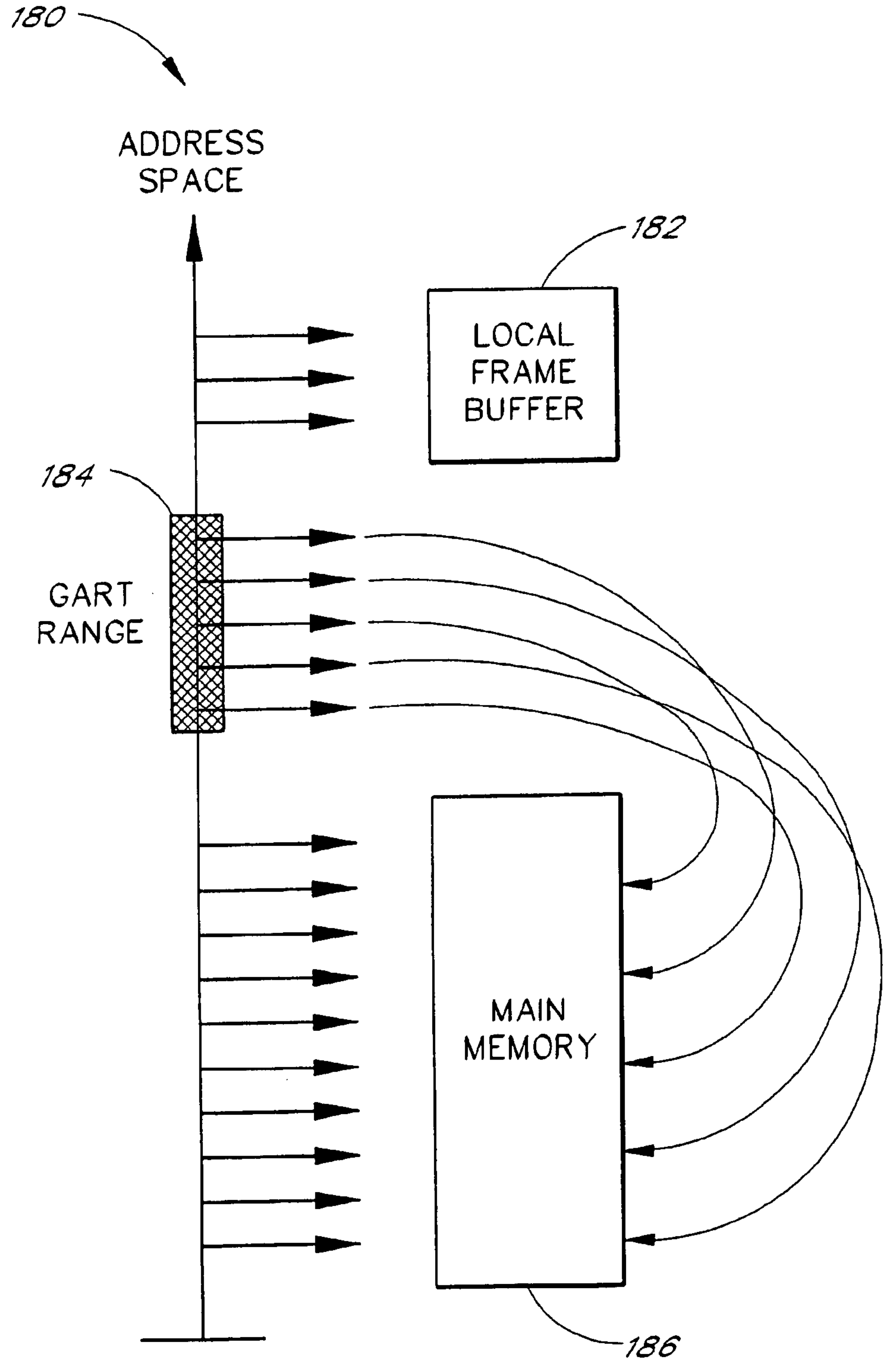

180
ADDRESS SPACE
182
LOCAL FRAME BUFFER
184
GART RANGE
MAIN MEMORY
186

FIG. 4

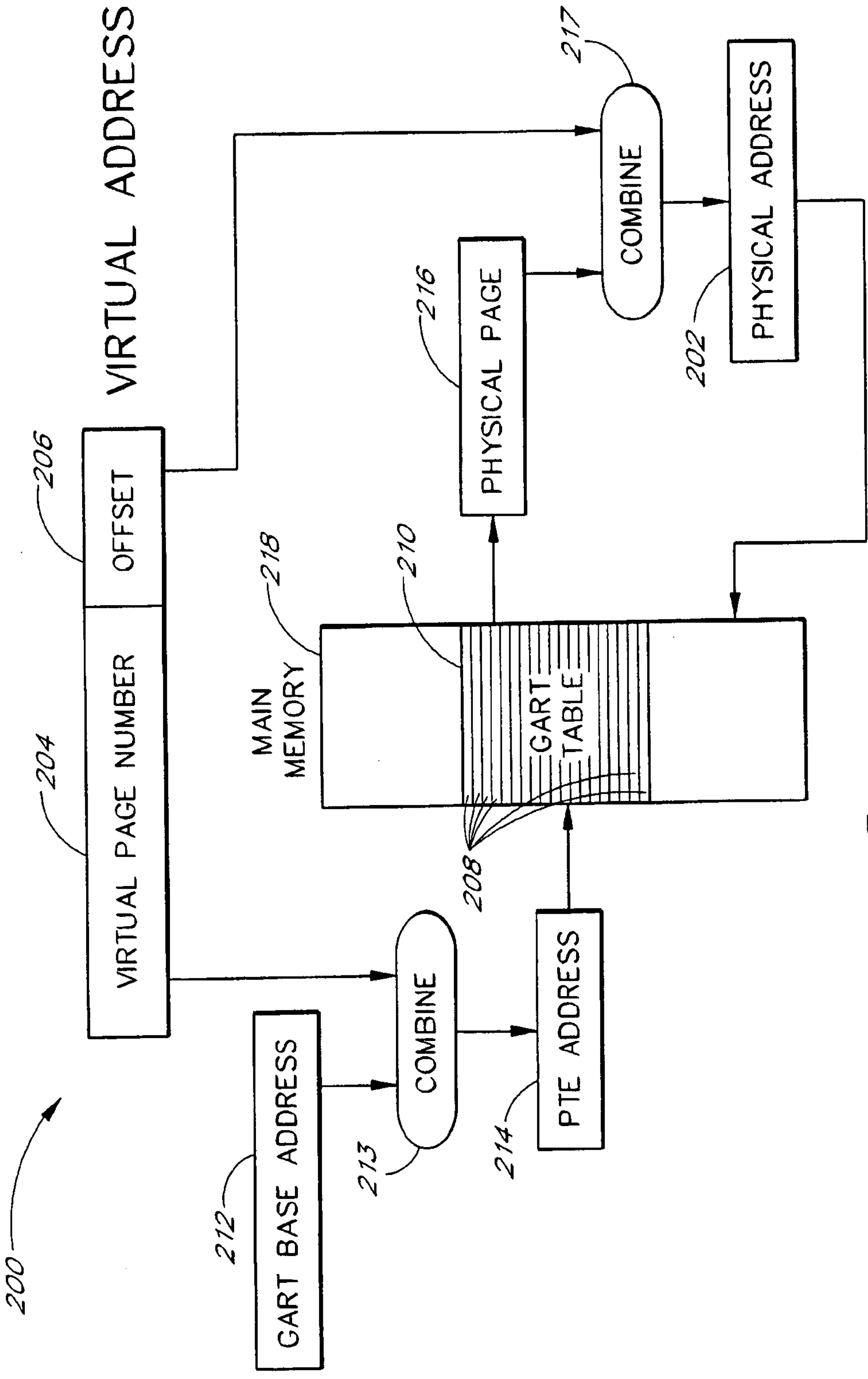

VIRTUAL ADDRESS
206
OFFSET
204
VIRTUAL PAGE NUMBER
217
COMBINE
PHYSICAL ADDRESS
202
216
PHYSICAL PAGE
218
MAIN MEMORY
210
GART TABLE
208
COMBINE
213
PTE ADDRESS
214
GART BASE ADDRESS
212
200

FIG. 5a

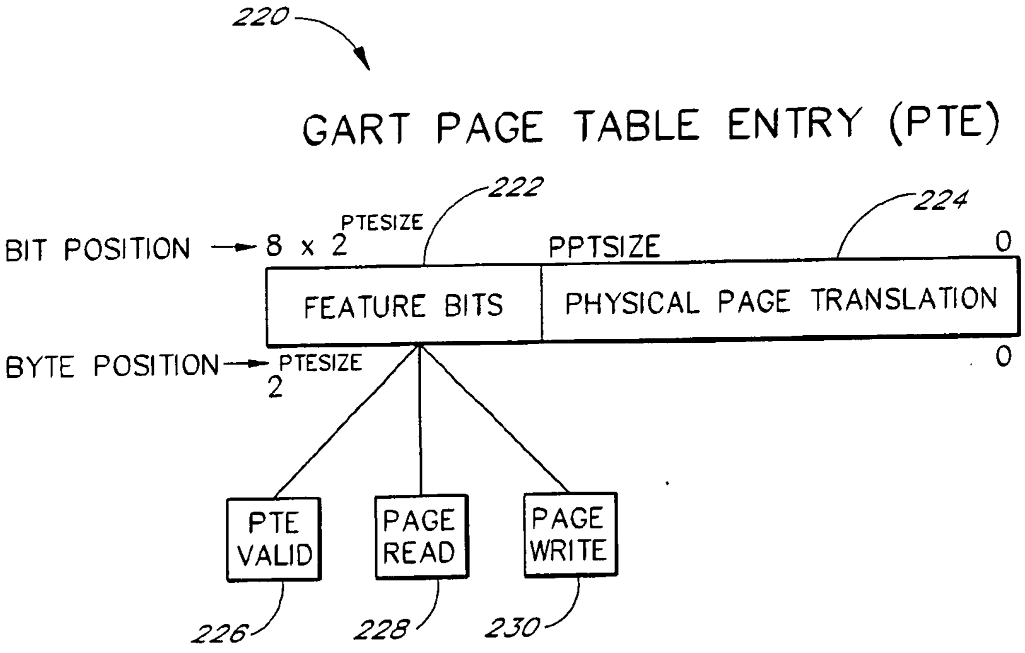

220
GART PAGE TABLE ENTRY (PTE)
222
224
BIT POSITION → 8 x 2^PTESIZE
PPTSIZE
0
FEATURE BITS
PHYSICAL PAGE TRANSLATION
BYTE POSITION → 2^PTESIZE
0
PTE VALID
PAGE READ
PAGE WRITE
226
228
230

FIG. 5b

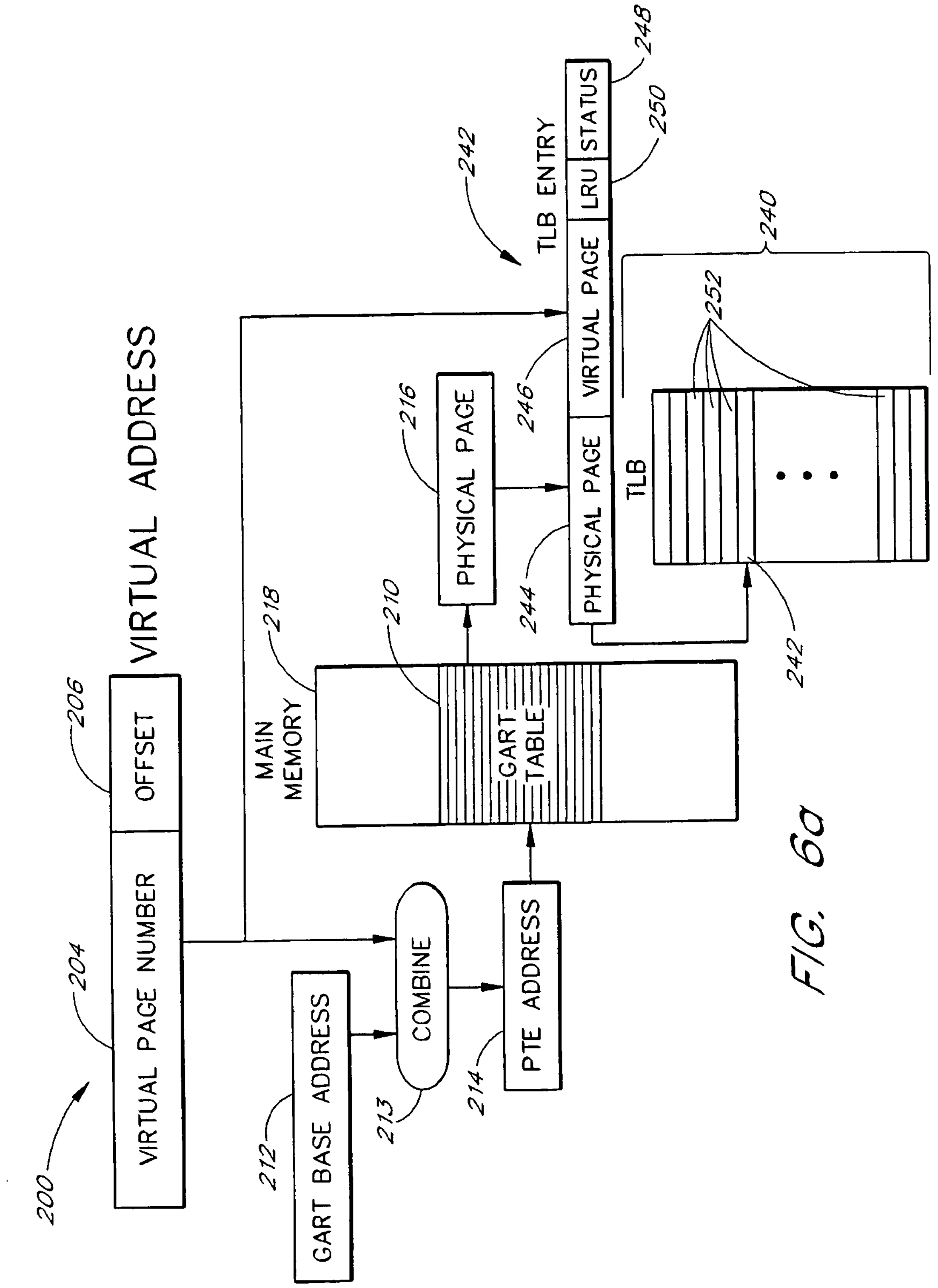

200
VIRTUAL ADDRESS
VIRTUAL PAGE NUMBER
OFFSET
204
206
GART BASE ADDRESS
212
COMBINE
213
PTE ADDRESS
214
MAIN MEMORY
218
GART TABLE
210
PHYSICAL PAGE
216
TLB ENTRY
242
PHYSICAL PAGE
VIRTUAL PAGE
LRU
STATUS
244
246
248
250
TLB
240
252
242

FIG. 6a

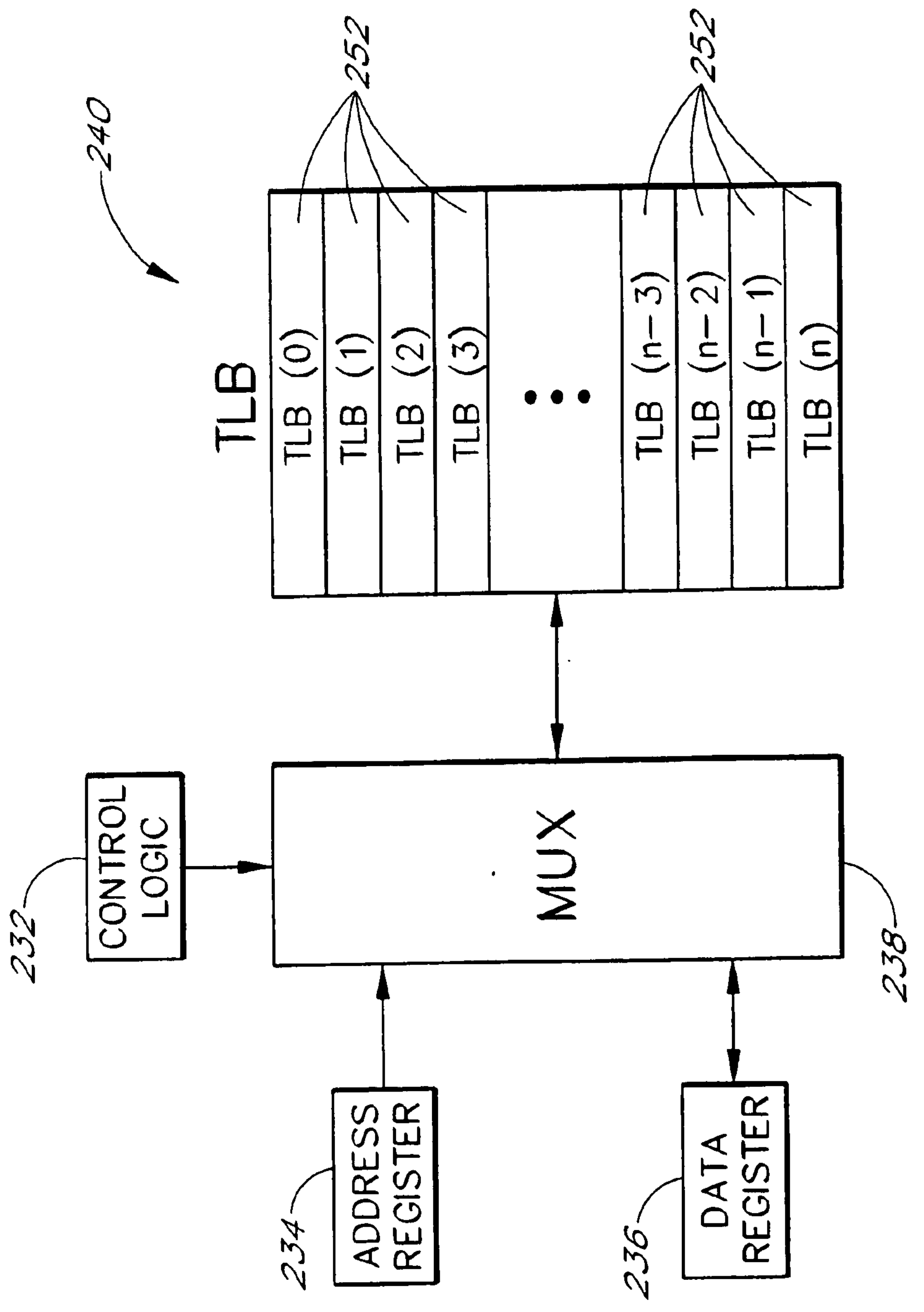

240
252
252
TLB
TLB (0)
TLB (1)
TLB (2)
TLB (3)
• • •
TLB (n–3)
TLB (n–2)
TLB (n–1)
TLB (n)
232
CONTROL LOGIC
MUX
238
234
ADDRESS REGISTER
236
DATA REGISTER

FIG. 6b

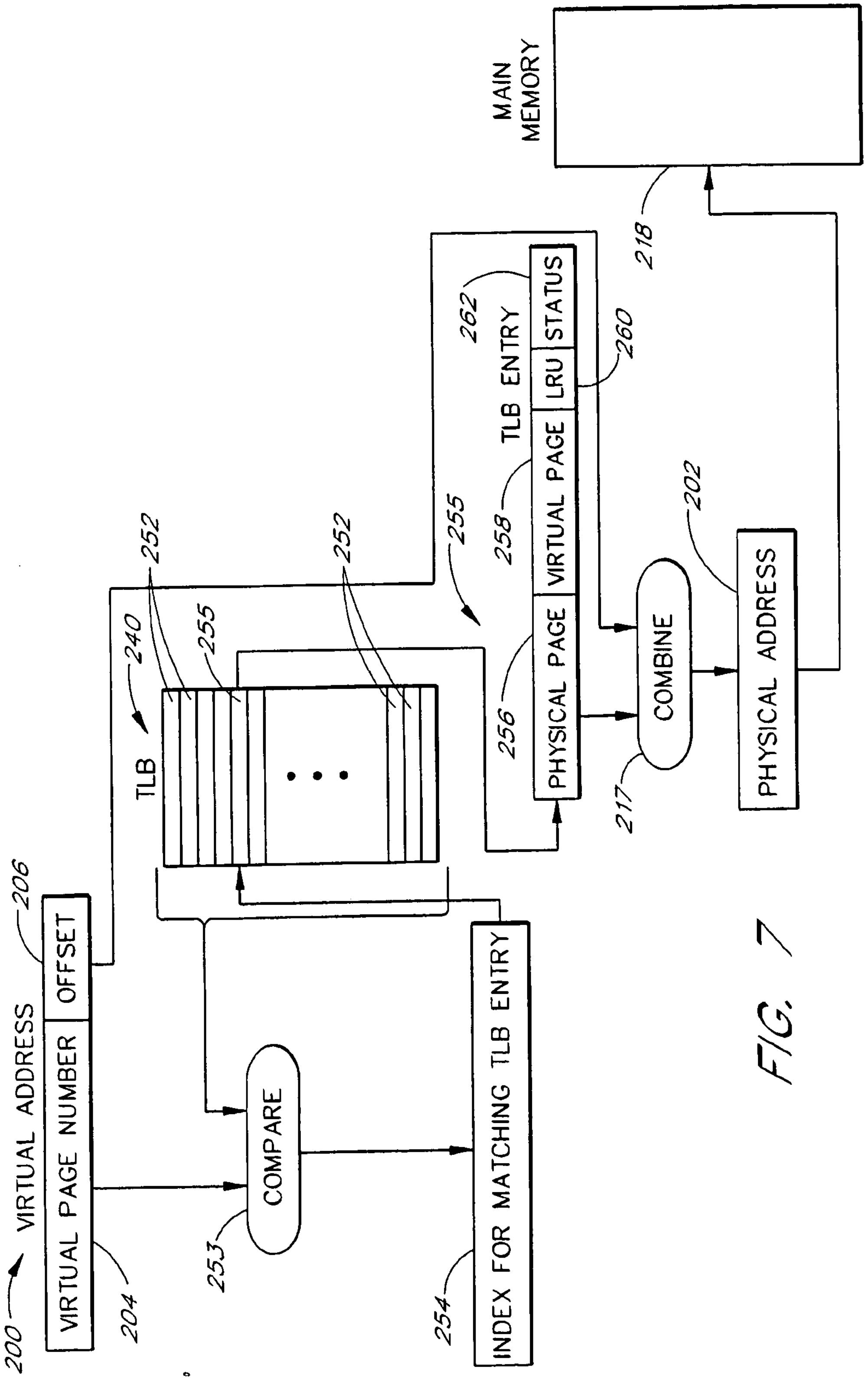

200
VIRTUAL ADDRESS
VIRTUAL PAGE NUMBER
OFFSET
204
206
COMPARE
253
INDEX FOR MATCHING TLB ENTRY
254
TLB
240
252
255
252
255
TLB ENTRY
PHYSICAL PAGE
256
VIRTUAL PAGE
258
LRU
260
STATUS
262
COMBINE
217
PHYSICAL ADDRESS
202
MAIN MEMORY
218

FIG. 7

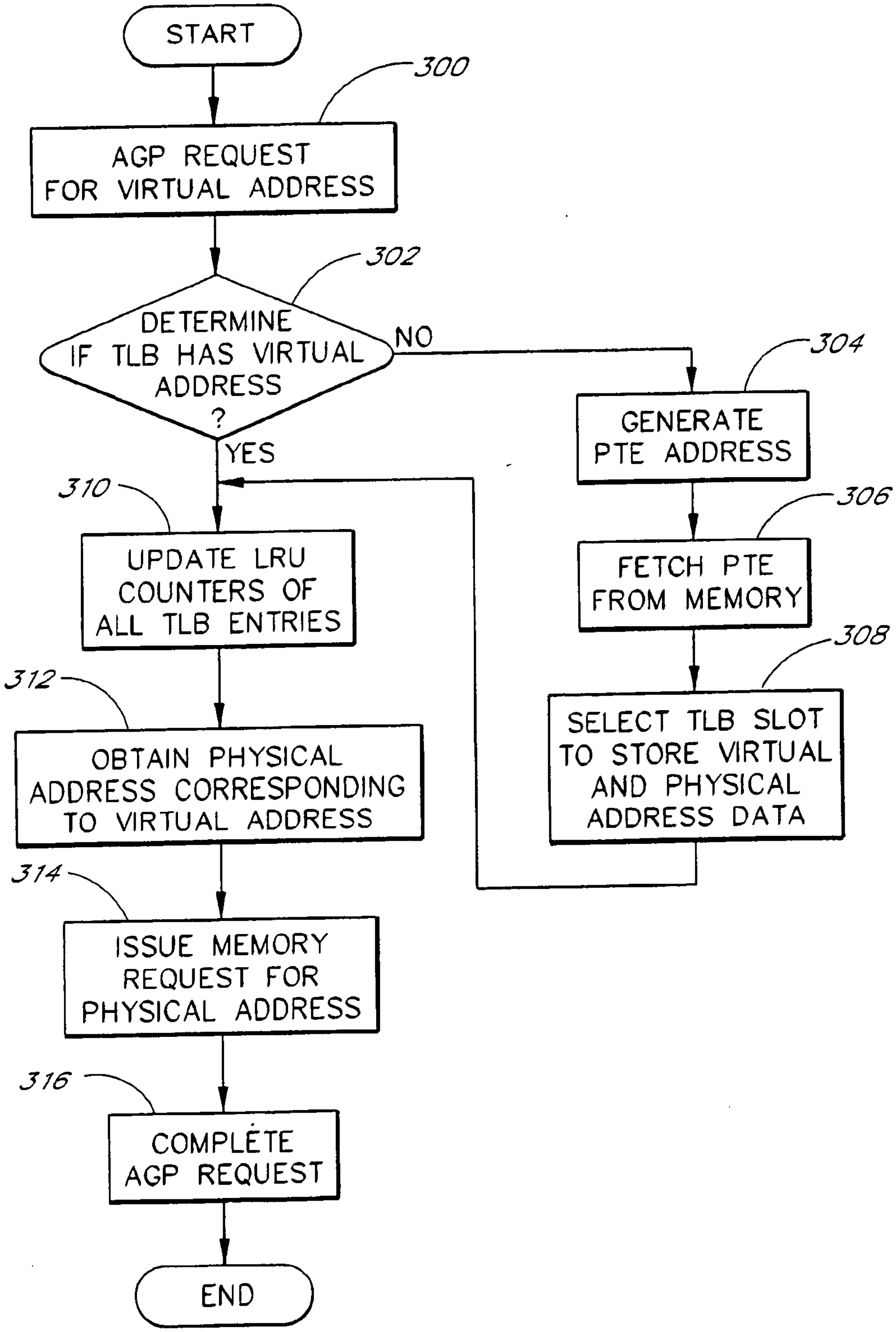

START
300
AGP REQUEST FOR VIRTUAL ADDRESS
302
DETERMINE IF TLB HAS VIRTUAL ADDRESS ?
NO
YES
304
GENERATE PTE ADDRESS
306
FETCH PTE FROM MEMORY
308
SELECT TLB SLOT TO STORE VIRTUAL AND PHYSICAL ADDRESS DATA
310
UPDATE LRU COUNTERS OF ALL TLB ENTRIES
312
OBTAIN PHYSICAL ADDRESS CORRESPONDING TO VIRTUAL ADDRESS
314
ISSUE MEMORY REQUEST FOR PHYSICAL ADDRESS
316
COMPLETE AGP REQUEST
END

FIG. 8

LRU UPDATE PROCESS

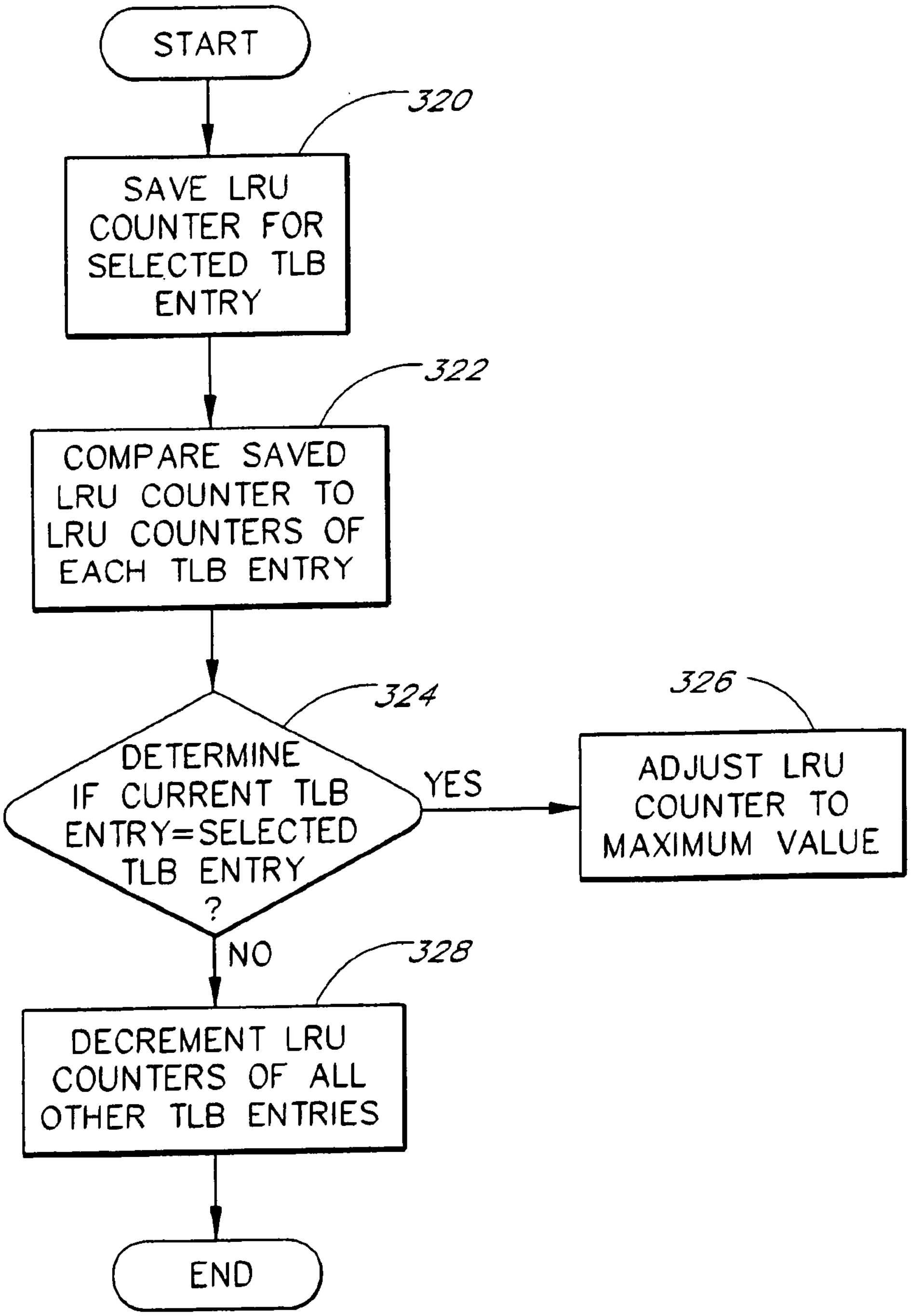

START
320
SAVE LRU COUNTER FOR SELECTED TLB ENTRY
322
COMPARE SAVED LRU COUNTER TO LRU COUNTERS OF EACH TLB ENTRY
324
DETERMINE IF CURRENT TLB ENTRY=SELECTED TLB ENTRY ?
YES
326
ADJUST LRU COUNTER TO MAXIMUM VALUE
NO
328
DECREMENT LRU COUNTERS OF ALL OTHER TLB ENTRIES
END

FIG. 9

TLB SELECTION PROCESS

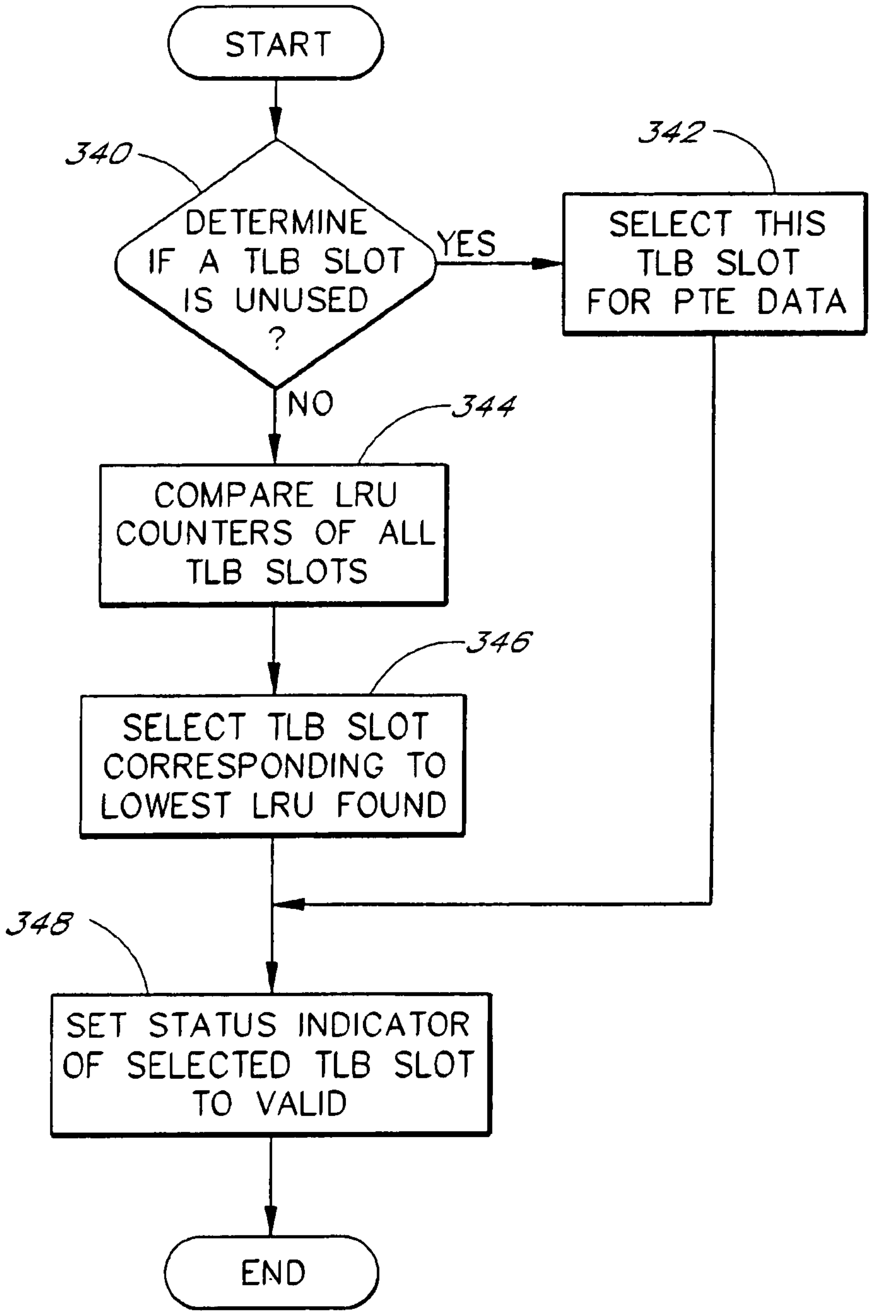

START
340
DETERMINE IF A TLB SLOT IS UNUSED ?
YES
342
SELECT THIS TLB SLOT FOR PTE DATA
NO
344
COMPARE LRU COUNTERS OF ALL TLB SLOTS
346
SELECT TLB SLOT CORRESPONDING TO LOWEST LRU FOUND
348
SET STATUS INDICATOR OF SELECTED TLB SLOT TO VALID
END

FIG. 10

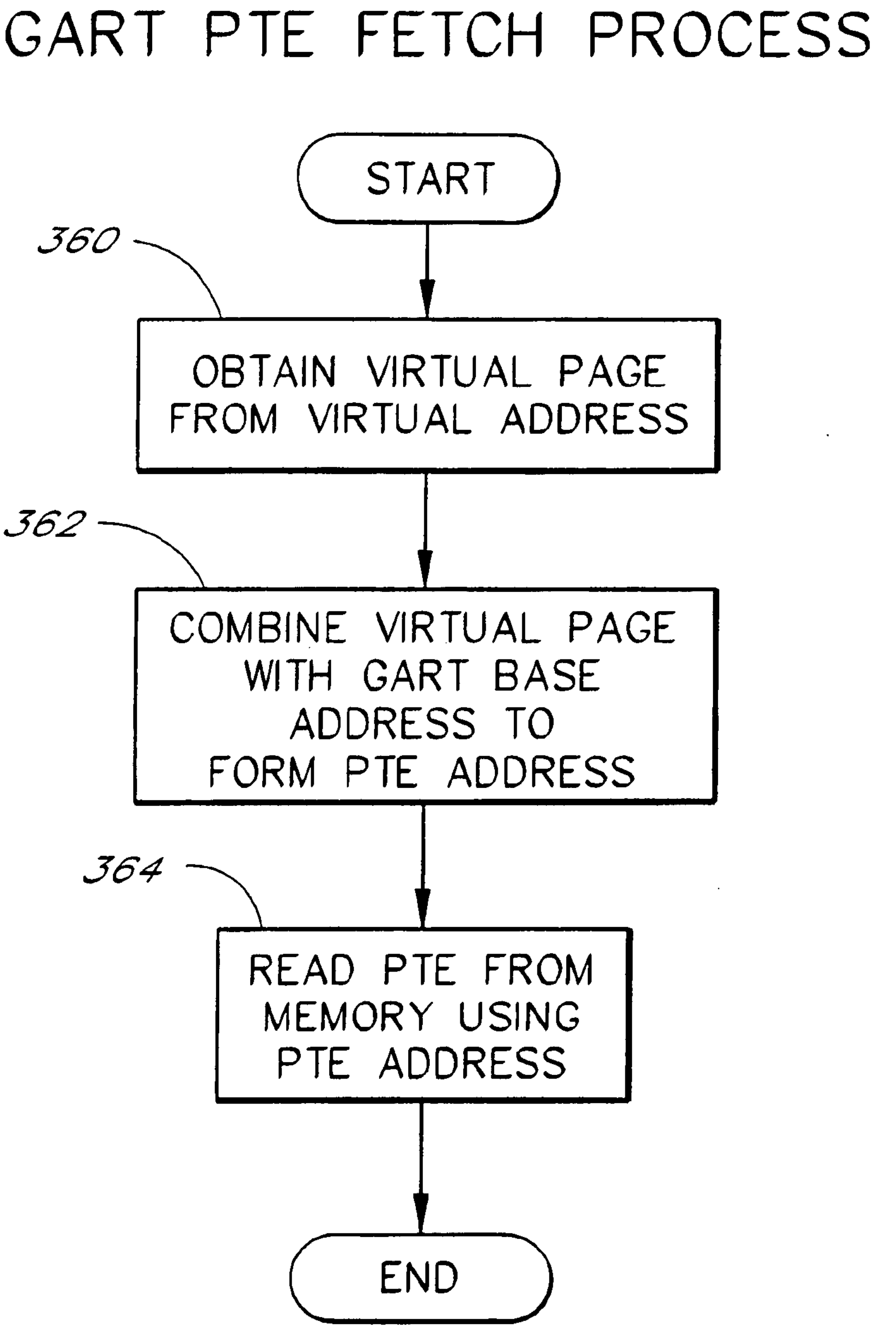

GART PTE FETCH PROCESS
START
360
OBTAIN VIRTUAL PAGE FROM VIRTUAL ADDRESS
362
COMBINE VIRTUAL PAGE WITH GART BASE ADDRESS TO FORM PTE ADDRESS
364
READ PTE FROM MEMORY USING PTE ADDRESS
END

FIG. 11

APPARATUS COMPRISING A TRANSLATION LOOKASIDE BUFFER FOR GRAPHICS ADDRESS REMAPPING OF VIRTUAL ADDRESSES

CROSS REFERENCE TO RELATED APPLICATIONS

This is a division of, and incorporates by reference in its entirety, U.S. application Ser. No. 08/882,054, now U.S. Pat. No. 6,249,853, titled "Apparatus for Graphic Address Remapping", filed Jun. 25, 1997. This application is related to, and incorporates by reference in their entirety, U.S. Pat. No. 6,069,638, filed Jun. 25, 1997, titled "System for Accelerated Graphics Port Address Remapping Interface to Main Memory", U.S. Pat. No. 6,282,625, filed Jun. 25, 1997, titled "Method for Accelerated Graphics Port Address Remapping Interface to Main Memory", U.S. application Ser. No. 09/723,403, filed Nov. 27, 2000, titled "Method for Implementing an Accelerated Graphics Port for a Multiple Memory Controller Computer System", and U.S. Pat. No. 6,252,612, filed Dec. 30, 1997, titled "Accelerated Graphics Port for Multiple Memory Controller Computer Systems".

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to computer systems, and more particularly, to an apparatus for mapping virtual addresses to physical addresses in graphics applications.

2. Description of the Related Technology

As shown in FIG. 1, a conventional computer system architecture 100 includes a processor 102, system logic 104, main memory 106, a system bus 108, a graphics accelerator 110 communicating with a local frame buffer 112 and a plurality of peripherals 114. The processor 102 communicates with main memory 106 through a memory management unit (MMU) in the system logic 104. Peripherals 114 and the graphics accelerator 110 communicate with main memory 106 and system logic 104 through the system bus 108. The standard system bus 108 is currently the Peripherals Connection Interface (PCI). The original personal computer bus, the Industry Standard Architecture (ISA), is capable of a peak data transfer rate of 8 megabytes/sec and is still used for low-bandwidth peripherals, such as audio. On the other hand, PCI supports multiple peripheral components and add-in cards at a peak bandwidth of 132 megabytes/sec. Thus, PCI is capable of supporting full motion video playback at 30 frames/sec, true color high-resolution graphics and 100 megabits/sec Ethernet local area networks. However, the emergence of high-bandwidth applications, such as three dimensional (3D) graphics applications, threatens to overload the PCI bus.

For example, a 3D graphics image is formed by taking a two dimensional image and applying, or mapping, it as a surface onto a 3D object. The major kinds of maps include texture maps, which deal with colors and textures, bump maps, which deal with physical surfaces, reflection maps, refraction maps and chrome maps. Moreover, to add realism to a scene, 3D graphics accelerators often employ a z-buffer for hidden line removal and for depth queuing, wherein an intensity value is used to modify the brightness of a pixel as a function of distance. A z-buffer memory can be as large or larger than the memory needed to store two dimensional images. The graphics accelerator 110 retrieves and manipulates image data from the local frame buffer 112, which is a type of expensive high performance memory. For example, to transfer an average 3D scene (polygon overlap of three) in 16-bit color at 30 frames/sec at 75 Hz screen refresh, estimated bandwidths of 370 megabytes/sec to 840 megabytes/sec are needed for screen resolutions from 640×480 resolution (VGA) to 1024×768 resolution (XGA). Thus, rendering of 3D graphics on a display requires a large amount of bandwidth between the graphics accelerator 110 and the local frame buffer 112, where 3D texture maps and z-buffer data typically reside.

In addition, many computer systems use virtual memory systems to permit the processor 102 to address more memory than is physically present in the main memory 106. A virtual memory system allows addressing of very large amounts of memory as though all of that memory were a part of the main memory of the computer system. A virtual memory system allows this even though actual main memory may consist of some substantially lesser amount of storage space than is addressable. For example, main memory may include sixteen megabytes (16,777,216 bytes) of random access memory while a virtual memory addressing system permits the addressing of four gigabytes (4,294,967,296 bytes) of memory.

Virtual memory systems provide this capability using a memory management unit (MMU) to translate virtual memory addresses into their corresponding physical memory addresses, where the desired information actually resides. A particular physical address holding desired information may reside in main memory or in mass storage, such as a tape drive or hard disk. If the physical address of the information is in main memory, the information is readily accessed and utilized. Otherwise, the information referenced by the physical address is in mass storage and the system transfers this information (usually in a block referred to as a page) to main memory for subsequent use. This transfer may require the swapping of other information out of main memory into mass storage in order to make room for the new information. If so, the MMU controls the swapping of information to mass storage.

Pages are the usual mechanism used for addressing information in a virtual memory system. Pages are numbered, and both physical and virtual addresses often include a page number and an offset into the page. Moreover, the physical offset and the virtual offset are typically the same. In order to translate between the virtual and physical addresses, a basic virtual memory system creates a series of lookup tables, called page tables, stored in main memory. These page tables store the virtual address page numbers used by the computer. Stored with each virtual address page number is the corresponding physical address page number which must be accessed to obtain the information. Often, the page tables are so large that they are paged themselves. The page number of any virtual address presented to the memory management unit is compared to the values stored in these tables in order to find a matching virtual address page number for use in retrieving the corresponding physical address page number.

There are often several levels of tables, and the comparison uses a substantial amount of system clock time. For example, to retrieve a physical page address using lookup tables stored in main memory, the typical MMU first looks to a register for the address of a base table which stores pointers to other levels of tables. The MMU retrieves this pointer from the base table and places it in another register. The MMU then uses this pointer to go to the next level of table. This process continues until the physical page address of the information sought is recovered. When the physical address is recovered, it is combined with the offset furnished as a part of the virtual address and the processor uses the result to access the particular information desired. Completion of a typical lookup in the page tables may take from ten to fifteen clock cycles at each level of the search.

To overcome this delay, virtual management systems often include cache memories called translation look aside buffers (TLBs). A TLB is essentially a buffer for caching recently translated virtual page addresses along with their corresponding physical page addresses. Such an address cache works on the same principle as do caches holding data and instructions, the most recently used addresses are more likely to be used than are other addresses. Thus, if a subsequent virtual address refers to the same page as the last one, the page table lookup process is skipped to save time. A TLB entry is like a cache entry wherein a tag portion includes portions of the virtual address and the data portion includes a physical page frame number, protections fields, use bits and status bits. When provided with a virtual page address stored in the TLB (a translation hit), the TLB furnishes a physical page address for the information without having to consult any page lookup tables. When the processor requests a virtual page address not stored in the TLB (a translation miss), the MMU must then consult the page lookup tables. When this occurs, the physical page address recovered is stored along with the virtual page address in the TLB so that it is immediately available for subsequent use. This saves a substantial amount of time on the next use of the information. For example, accessing the information using a TLB may require only one or two clock cycles compared to the hundreds of clock cycles required for a page table lookup.

Virtual memory systems are common in the art. For example, in U.S. Pat. No. 5,446,854, Khalidi et al. disclose a method and apparatus for virtual to physical address translation using hashing. Similarly, Crawford et al. disclose a microprocessor architecture having segmentation mechanisms for translating virtual addresses to physical addresses in U.S. Pat. No. 5,321,836. Lastly, in U.S. Pat. Nos. 5,491,806 and 5,546,555, Horstmann, et al. disclose an optimized translation lookaside buffer for use in a virtual memory system.

As shown in FIG. 1, moving 3D graphics data to the main memory **106** in current computer systems would require the graphics accelerator **110** to access the 3D graphics data through the PCI system bus **108**. Thus, although Bechtolsheim discloses a data bus enabling virtual memory data transfers in U.S. Pat. Nos. 4,937,734 and 5,121,487, 3D rendering exceeds the peak PCI bandwidth of 132 megabytes/sec because a bandwidth of at least 370 megabytes/sec is needed for data transfer from main memory **106**. Moreover, the graphics accelerator **110** often requires storage of graphics data into large contiguous blocks of memory. For example, a 16-bit 256×256 pixel texture map for 3D graphics applications requires a memory block having a size of 128K bytes. However, operating system software, such as Microsoft®, Windows®, Windows® 95 and Windows NT®, and the system logic **104** often allocate main memory in page frames having smaller sizes, such as 4K. In U.S. Pat. No. 5,465,337, Kong discloses a memory management unit capable of handling virtual address translations for multiple page sizes. However, this does not address the bandwidth limitations of the PCI bus discussed above. In order to move 3D graphics data from the local frame buffer **112** to main memory **106**, computer systems require an improved method for storing and addressing graphics data in main memory.

In U.S. Pat. No. 5,313,577, Meinerth et al. discloses a graphics processor capable of reading from, and writing to, virtual memory. This graphics processor can be described by reference to FIG. 2, which illustrates a graphics/memory control unit **120** including a graphics processor unit **122** that communicates with a memory control unit **124**. The graphics/memory control unit **120** in turn communicates with the main memory **106** and the frame buffer **112** through a dedicated memory bus **126**. The graphics processor unit **122** includes an address generator and a virtual translation unit to provide for translation of virtual addresses to physical addresses when accessing the main memory **106** and the frame buffer **112**. In addition, the memory control unit **124** communicates with a processor **102** through a dedicated system bus **128**, with an I/O device **114** through a dedicated I/O bus **130** and with computer networks through a dedicated network bus **132**. In contrast to the structure of FIG. 1, the use of dedicated buses for communication with the main memory **106**, I/O devices **114** and computer networks substantially increases system cost and decreases the flexibility with which a computer system can be upgraded. For example, to upgrade the graphics capability of a computer system having the structure as illustrated in FIG. 1, one simply connects a more powerful graphics adapter to the PCI bus **108** (FIG. 1). However, upgrading the graphics capability of a computer system having the structure of FIG. 2 requires replacement of the memory control unit **124** as well as the graphics processor unit **122**. Similarly, the structure of FIG. 2 is not compatible with the vast majority of available PCI enhancement devices. Moreover, the structure of FIG. 2 also requires the graphics processor unit **122** to access 3D graphics data through a memory bus **126**.

In view of the limitations discussed above, computer manufacturers require a modular architecture that reduces the cost of system upgrades, such as enhanced 3D graphics adapters, to improve display performance. Similarly, to reduce system memory costs, computer manufacturers require improved methods for storing, addressing and retrieving graphics data from main memory instead of expensive local frame buffer memory. Moreover, to address the needs of high bandwidth graphics applications without substantial increases in system cost, computer manufacturers require improved technology to overcome current system bus bandwidth limitations.

SUMMARY OF THE INVENTION

One embodiment of the invention includes a graphics address remapping table (GART), the GART stored in memory, comprising at least one page table entry (PTE) providing information for translation of a virtual address to a physical address, wherein the virtual address includes a first portion and a second portion, the first portion being used to locate a PTE in the GART corresponding to the virtual address and wherein the second portion and the information provided by the PTE are combined to provide the physical address.

Another embodiment of the invention includes a page table entry for a graphics address remapping table stored in memory comprising a physical page translation field having translation information and a feature bits field having at least one indicator defining an attribute of the physical page translation field.

Yet another embodiment of the invention includes a translation lookaside buffer (TLB) in a memory, the TLB receiving a portion of a virtual address selected from a graphics address remapping range, comprising at least one TLB entry, wherein each of the at least one TLB entries includes a virtual page field and a corresponding physical page field, wherein if the portion of the virtual address matches the virtual page field of one TLB entry, the TLB provides translation information from the physical page field of the one TLB entry to form a physical address.

Yet another embodiment of the invention includes an apparatus for graphic address remapping of a virtual address comprising a graphics address remapping table (GART) stored in memory and having information which is used to translate the virtual address to a physical address and a translation lookaside buffer (TLB) receiving a portion of the virtual address, the TLB having at least one TLB entry, wherein each of the at least one TLB entries includes a virtual page field and a corresponding physical page field, wherein if the portion of the virtual address matches the virtual page field of one TLB entry, the TLB provides translation information from the physical page field of the one TLB entry to form the physical address and wherein if the portion of the virtual address does not match the virtual page field of one TLB entry, the GART provides translation information referenced by the portion of the virtual address to form the physical address.

Lastly, yet another embodiment of the present invention includes an apparatus for graphic address remapping of a virtual address comprising an interface and a translation lookaside buffer (TLB) in communication with the interface, the TLB having at least one TLB entry including information which is used to translate the virtual address to a physical address, wherein the interface receives a portion of the virtual address and provides access to a TLB entry corresponding to the portion of the virtual address.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram illustrating the architecture of a prior art computer system.

FIG. **2** is a block diagram illustrating the architecture of another prior art computer system.

FIG. **3** is a block diagram illustrating the architecture of a computer system of one embodiment of the present invention.

FIG. **4** is a diagram illustrating the address space of a processor of one embodiment of the present invention.

FIG. **5***a* is a diagram illustrating the translation of a virtual address to a physical address of one embodiment of the present invention.

FIG. **5***b* is a diagram illustrating a page table entry (PTE) of the graphic address remapping table (GART) of one embodiment of the present invention.

FIG. **6***a* is a diagram illustrating the generation of a translation look aside buffer (TLB) entry of one embodiment of the present invention.

FIG. **6***b* is a block diagram illustrating one embodiment of an interface for the direct access of a translation look aside buffer (TLB) of one embodiment of the present invention.

FIG. **7** is a diagram illustrating the translation of a virtual address to a physical address using the TLB of one embodiment of the present invention.

FIG. **8** is a flowchart illustrating the method of processing an AGP request of the present invention.

FIG. **9** is a flowchart illustrating the method of updating a least recently used (LRU) counter of one embodiment of the present invention.

FIG. **10** is a flowchart illustrating the method of selecting a slot to store a TLB entry of one embodiment of the present invention.

FIG. **11** is a flowchart illustrating the method of fetching a page table entry (PTE) of one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

In contrast to the conventional computer system architecture **100** (FIG. **1**), embodiments of the present invention enables relocation of a portion of the 3D graphics data, such as the texture data, from the local frame buffer **112** (FIG. 1) to main memory **106** (FIG. 1) to reduce the size, and thus the cost, of the local frame buffer **112** and to improve system performance. For example, as texture data is generally read only, moving it to main memory **106** does not cause coherency or data consistency problems. Similarly, as texture data is loaded from mass storage into main memory **106**, leaving it in main memory **106** instead of copying it to the local frame buffer **112** reduces overhead. Moreover, as the complexity and quality of 3D images has increased, leaving 3D graphics data in the local frame buffer **112** has served to increase the computer system cost over time. Thus, although moving 3D graphics data to main memory **106** may likewise require an increase in the size of the main memory **106**, the architecture of the present invention reduces the total system cost because it is less expensive to increase main memory **106** than to increase local frame buffer memory **112**.

Referring now to FIG. **3**, the computer system architecture **150** of one embodiment of the present invention includes a processor **152**, system logic **154**, main memory **156**, a system bus **158**, a graphics accelerator **160** communicating with a local frame buffer **162** and a plurality of peripherals **164**. The processor **152** communicates with the main memory **156** through a memory management unit (MMU) in the system logic **154**. Peripherals **114** communicate with the main memory **156** and system logic **154** through the system bus **158**. Note however that the graphics accelerator **160** communicates with the system logic **154** and main memory **156** through an accelerated graphics port (AGP) **166**. AGP **166** is not a bus, but a point-to-point connection between an AGP compliant target, the MMU portion of the system logic **154**, and an AGP-compliant master, the graphics accelerator **160**. The AGP **166** point-to-point connection enables data transfer on both the rising and falling clock edges, improves data integrity, simplifies AGP protocols and eliminates bus arbitration overhead. AGP provides a protocol enhancement enabling pipelining for read and write accesses to main memory **156**.

For example, the graphics accelerator **160** initiates a pipelined transaction with an access request. System logic **154** responds to the request by initiating the corresponding data transfer at a later time. The graphics accelerator **160** can then issue its next pipelined transaction while waiting for the previous data to return. This overlap results in several read or write requests being active at any point in time. In one embodiment, the AGP **166** operates with a 66 MHz clock using 3.3 volt signaling. As data transfers can occur on both clock edges, the AGP **166** enables an effective 133 MHz data transfer rate and can reach a peak bandwidth of 533 megabytes/sec. For detailed information on the AGP **166**, consult the Accelerated Graphics Port Interface Specification, Revision 1.0, released by Intel Corporation and available from Intel in Adobe¤ Acrobat¤ format on the World Wide Web . This document is hereby incorporated by reference.

As noted above, the embodiment of FIG. **3** enables the graphics accelerator **160** to access both main memory **156** and the local frame buffer **162**. From the perspective of the graphics accelerator **160**, the main memory **156** and the local frame buffer **162** are logically equivalent. Thus, to optimize system performance, graphics data may be stored in either the main memory **156** or the local frame buffer **162**. In contrast to the direct memory access (DMA) model where graphics data is copied from the main memory **156** into the local frame buffer **162** by a long sequential block transfer prior to use, the graphics accelerator **160** of the present invention can also use, or "execute," graphics data directly from the memory in which it resides (the "execute" model). However, since the main memory **156** is dynamically allocated in random pages of a selected size, such as 4K, the "execute" model requires an address mapping mechanism to map random pages into a single contiguous, physical address space needed by the graphics accelerator **160**.

FIG. **4** illustrates an embodiment of the address space **180** of the computer system **150** (FIG. **3**) of the present invention. For example, a **32** bit processor **152** (FIG. **3**) has an address space **180** including 2^{32} (or 4,294,967,296) different addresses. A computer system **150** (FIG. **3**) typically uses different ranges of the address space **180** for different devices and system agents. In one embodiment, the address space **180** includes a local frame buffer range **182**, a graphics address remapping table (GART) range **184** and a main memory range **186**. In contrast to prior art systems, addresses falling within the GART range **184** are remapped to non-contiguous pages within the main memory range **186**. All addresses not in the GART range **184** are passed through without modification so that they map directly to the main memory range **186** or to device specific ranges, such as the local frame buffer range **182**. In one embodiment, the system logic **154** performs the address remapping using a memory based table, the GART, defined in software with an application program interface (API). Moreover, the GART table format is abstracted to the API by a hardware abstraction layer (HAL) or a miniport driver provided by the system logic **154**. Thus, by defining the GART in software, the present invention advantageously provides the substantial implementation flexibility needed to address future partitioning and remapping circuitry (hardware) as well as any current or future compatibility issues.

FIG. **5***a* illustrates the translation of a virtual address **200** to a physical address **202** in one embodiment of the present invention. As discussed previously, in one embodiment, only those virtual addresses falling within the GART range **184** (FIG. **4**) are remapped to main memory **186** (FIG. **4**). A virtual address **200** includes a virtual page number field **204** and an offset field **206**. Translation of the contents of the virtual page number field **204** occurs by finding a page table entry (PTE) corresponding to the virtual page number field **204** among the plurality of GART PTEs **208** in the GART table **210**. To identify the appropriate PTE having the physical address translation, the GART base address **212** is combined at **213** with the contents of the virtual page number field **204** to obtain a PTE address **214**. The contents referenced by the PTE address **214** provide the physical page number **216** corresponding to the virtual page number **204**. The physical page number **216** is then combined at **217** with the contents of the offset field **206** to form the physical address **202**. The physical address **202** in turn references a location in main memory **218** having the desired information.

The GART table **210** may include a plurality of PTEs **208** having a size corresponding to the memory page size used by the processor **152** (FIG. **3**). For example, an Intel¤ Pentium¤ or Pentium¤ Pro processor operates on memory pages having a size of 4K. Thus, a GART table **210** adapted for use with these processors may include PTEs referencing 4K pages. In one embodiment, the virtual page number field **204** comprises the upper 20 bits and the offset field **206** comprises the lower 12 bits of a 32 bit virtual address **200**. Thus, each page includes 2^{12}=4096 (4K) addresses and the lower 12 bits of the offset field **206** locate the desired information within a page referenced by the upper 20 bits of the virtual page number field **204**. The GART table **210** preferably resides in the main memory **218**. Memory refers generally to storage devices, such as registers, SRAM, DRAM, flash memory, magnetic storage devices, optical storage devices and other forms of volatile and non-volatile storage.

FIG. **5***b* illustrates one possible format for a GART PTE **220**. The GART PTE **220** includes a feature bits field **222** and a physical page translation (PPT) field **224**. In contrast to prior art systems where hardwired circuitry defines a page table format, the GART table **210** (FIG. **5***a*) may include PTEs of configurable length enabling optimization of table size and the use of feature bits defined by software. The length of the GART PTE **220** is $2^{PTESize}$ bytes or $8*2^{PTESize}$ bits. For example, for a PTESize=5, the GART PTE has a length of 32 bytes or 256 bits. The PPT field **224** includes PPTSize bits to generate a physical address **202** (FIG. **5***a*). PPTSize defines the number of translatable addresses, and hence the GART table **210** (FIG. **5***a*) includes $2^{PPTSize}$ PTE entries. As PTESize defines the size of each GART PTE **220**, the memory space needed for the entire GART table **210** (FIG. **5***a*) is $2^{(PTESize+PPTSize)}$ bytes. For example, the GART table **210** in a system with a 4K (=2^{12}) memory page size and 32 megabytes (=2^{25}) of main memory **218** (FIG. **5***a*) includes $2^{25}/2^{12}=2^{13}$=8192 PTEs. Thus, only 13 bits are needed to define 8192 unique PTEs to span the entire 32 megabytes of main memory **218** (FIG. **5***a*) and PPTSize=13. However, to accommodate various software feature bits, each PTE may have a size of 8 bytes (=2^3 and PTESize=3). Thus, the size of the GART table **210** is $2^{(PTESize+PPTSize)}=2^{(3+13)}=2^{16}$=65536 bytes=64K.

As noted above, the GART table **210** (FIG. **5***a*) may use 4K page boundaries. Thus, when (PTESize+PPTSize) is less than 12 bits (2^{12}=4096 bytes=4K), the entire GART table **210** (FIG. **5***a*) resides within one 4K page. For values greater than 12, the GART table **210** (FIG. **5***a*) resides on multiple 4K pages. To maintain compatibility with the Intel¤ Pentium¤ Pro processor caches, the GART base address **214** (FIG. **5***a*) may begin on a $2^{(PTESize+PPTSize)}$ byte boundary. Thus, a GART base address **214** (FIG. **5***a*) can not have a value which aligns the GART table **210** (FIG. **5***a*) on an address boundary less than the size of the GART table **210** (FIG. **5***a*). For example, an 8K GART table **210** (FIG. **5***a*) must begin on a 8K boundary.

In one embodiment, an initialization BIOS implements the GART table **210** (FIG. **5***a*) by loading configuration registers in the system logic **154** (FIG. **3**) during system boot up. In another embodiment, the operating system implements the GART table **210** (FIG. **5***a*) using an API to load the configuration registers in the system logic **154** (FIG. **3**) during system boot up. The operating system then determines the physical location of the GART table **210** (FIG. **5***a*) within main memory **218** (FIG. **5***a*) by selecting the proper page boundary as described above (i.e., an 8K GART table begins on an 8K boundary). For example, the system loads configuration registers holding the GART base address **214** (FIG. **5***a*) defining the beginning of the GART table **210** (FIG. **5***a*), PTESize defining the size of a GART PTE **220** and PPTSize defining the size of the physical address used to translate a virtual address. In addition, the system loads a configuration register forAGPAperture, defining the lowest address of the GART range **184** (FIG. **4**), and PhysBase, defining the remaining bits needed to translate a virtual address not included in the PPTSize bits.

For example, consider a system having 64 megabytes of main memory **218** (FIG. **5***a*) encompassing physical addresses 0 through 0x03FFFFFF with the AGP related data occupying the upper 32 megabytes of main memory **218** referenced by physical addresses 0x02000000 through 0x03FFFFFF. If the GART Range **184** (FIG. **4**) begins at the 256 megabyte virtual address boundary 0x10000000, the invention enables translation of virtual addresses within the GART Range **184** to physical addresses in the upper 32 megabytes of main memory **218** corresponding to physical addresses in the range 0x02000000 through 0x03FFFFFF. As noted earlier, a GART table **210** includes multiple PTEs, each having physical page translation information **224** and software feature bits **222**. The GART table **210** may be located at any physical address in the main memory **218**, such as the 2 megabyte physical address 0x00200000. For a system having a 4K memory page size and a GART PTE **220** of 8 byte length, the GART table **210** is configured as follows:

PhysBase	:=0×02000000	—Start of remapped physical address
PhysSize	:=32 megabytes	—Size of remapped physical addresses
AGPAperture	:=0×10000000	—Start address of GART Range
GARTBase	:=0×00200000	—Start address of GART table
$2^{PTESize}$	:=8 bytes	—Size of each GART Page Table Entry
PageSize	:=4 kilobytes	—Memory page size

To determine the number of PTEs in the GART table **210**, the size of the physical address space in main memory **218** allocated to AGP related data, the upper 32 megabytes=33554432 bytes, is divided by the memory page size, 4K=4096 bytes, to obtain 8192 PTEs. Note that $8192=2^{13}=2^{PTESize}$ and thus, PTESize=13. To implement the GART table **210**, the configuration registers are programmed with the following values:

PhysBase	:=0×02000000	—Start of remapped physical address
AGPAperture	:=0×10000000	—Start address of GART Range
GARTBase	:=0×00200000	—Start address of GART table
PTESize	:=3	—Size of each GART PTE
PPTSize	:=13	—Number of PPT bits in each PTE

Lastly, the GART table **210** is initialized for subsequent use.

Using pseudo-VHDL code, system logic **154** (FIG. **3**) can quickly determine whether a 32 bit AGP address (AGPAddr) requires translation from a virtual to physical address (PhysAddr) as follows:

```
if ((AGPAddr(31 downto 12) and not ($2^{PPTSize\ -1)}$)=
    AGPAperture (31 downto 12)) then
    Virtual=true;
else
    Virtual=false;
end if;
```

When the address is virtual, then the PTE address 214 (PTEAddr) is calculated as follows:

PTEAddr<=((AGPAddr(31 downto 12) and ($2^{(PPTSize)}$–1)) shl $2^{PTESize}$) or (GARTBase and not ($2^{(PTESize+PPTSize)}$–1)));

Note that the "shl" function indicates a left shift with zero fill, which can be implemented in hardware using a multiplexer. Lastly, to determine the physical address **202** (PhysAddr) when PPTSize does not include sufficient bits to remap the entire GART range **184** (FIG. **4**), the physical page **216** is generated as follows:

PhysAddr(31 downto 12)<=(PhysBase(31 downto 12) and not ($2^{PPTSize}$–1)) or (PTE and ($2^{PPTSize}$–1)));

To obtain the physical address **202**, the physical page **216**, PhysAddr(31 downto 12), is then combined with the offset **206**. Note that the pseudo-code above avoids the use of adders, which impede system performance at high clock frequencies, in the virtual to physical address translation process.

To illustrate the use of the pseudo-code above, suppose an AGP master, such as the graphics accelerator **160** (FIG. **3**), presents the virtual address 0x0002030, which corresponds to AGPAddr in the pseudo-code, to the system logic **154** (FIG. **3**) for translation. To determine if AGPAddr=0x10002030 is appropriate for translation using the GART table configured above, the system logic **154** first evaluates the if condition:

((AGPAddr(31 downto 12) and not ($2^{PPTSize}$–1))=AGPAperture (31 downto 12))

to determine if it is true or false. In addition, the expression ($2^{PPTSize}$–1) indicates that the lower PPTSize bits are set, which is easily performed in hardware. For the GART table **210** configured above, note that PPTSize=13, ($2^{PPTSize}$–1)=0x01IFFF (hexadecimal) and AGPAperture=0x10000000. The notation (31 downto 12) indicates use of bit positions 12 through 31 of an address, which is equivalent to truncating the lower 12 bits of a binary address or the lower three values of a hexadecimal address. Thus, for AGPAddr=0x10002030 and AGPAperture=0x10000000, AGPAddr(31 downto 12)=0x10002 and AGPAperture(31 downto 12)=0x10000. Now, substitute the values for AGPAddr, AGPAperture and ($2^{PPTSize}$–1) into the if condition:

((AGPAddr(31 downto 12) and not ($2^{PPTSize}$–1))=AGPAperture (31 downto 12)) -or-

(0x10002 and not (0x01FFF))=0x10000 -or-

0x10000=0x10000

to calculate a result. Here, the result is true indicating that AGPAddr=0x10002030 is a valid address for translation. Similarly, for the virtual address 0x11002030, the if condition produces this result: 0x11000=0x10000. As 0x110000x10000, this result is false indicating that the virtual address 0x11002030 does not fall within the GART range **184**. If an AGP master presented the virtual address 0x11002030, the system logic **154** reports an error.

To determine the location of the PTE in the GART table **210** having the translation information for the virtual address AGPAddr=0x10002030, the expression:

PTEAddr<=((AGPAddr(31 downto 12) and ($2^{(PPTSize)}$–1)) shl $2^{PTESize}$) or (GARTBase and not ($2^{(PTESize+PPTSize)}$–1)))

is evaluated. For the GART table **210** configured above, GARTBase=0x00200000, PPTSize=13, PTESize=3 and ($2^{(PTESize+PPTSize)}$–1)=0x0FFFF. As noted above, ($2^{PPTSize}$–1)=0x01FFF and AGPAddr(31 downto 12)=0x10002. Now, substitute the values into the equation for PTEAddress:

PTEAddr<=((0x10002 and 0x01FFF) shl 3) or (0x00200000 and not (0x0FFFF)) -or-

PTEAddr<=(0x00002 shl 3) or (0x00200000) -or-

PTEAddr<=(0x00000010) or (0x00200000)=0x00200010.

As each PTE occupies 8 bytes and the GART table **210** begins at the GARTBase address=0x00200000, the calculated PTEAddress=0x00200010 corresponds to the third entry or PTE(2), 16 bytes away from the GARTBase address. Suppose that the lower 32 bits (or 4 bytes) of the value at PTE(2)=0x12345678. As shown in the embodiment of FIG. 5*b*, the lower PPTSize=13 bits correspond to the PPT translation bits and the higher order bits are software feature bits 222. Of course, in another embodiment, the PPT translation information may comprise the higher order bits while the software feature bits **222** may comprise the lower order bits. Moreover, the PPT translation information and the software feature bits **222** may be located at any of the bit positions within a PTE **220**.

Lastly, to calculate the physical address corresponding to the virtual address AGPAddr=0x10002030, the expression:

PhysAddr(31 downto 12)<=(PhysBase(31 downto 12) and not ($2^{PPTSize}$-1)) or (PTE and ($2^{PPTSize}$-1)))

is evaluated. For the GART table **210** configured above, PhysBase=0x02000000 and ($2^{PPTSize}$-1)=0x01FFF. Note also that PTE(2)=0x12345678. Now, substitute the values into the equation for PhysAddr(31 downto 12):

PhysAddr(31 downto 12)<=(0x02000 and not (0x01FFF)) or (0x12345678 and 0x01FFF)) -or-

PhysAddr(31 downto 12)<=(0x02000) or (0x00001678)= 0x03678. Note that the offset 206 corresponds to the lower 12 bits of the virtual address 0x10002030 or AGPAddr(11 downto 0)=030. Thus, to obtain the physical address **206**, the physical page **216** is combined with the offset **206** to form PhysAddr(31 downto 0) or 0x03678030. To summarize, the pseudo-code of the embodiment described above illustrates the translation of the virtual address 0x10002030 to the physical address 0x03678030.

Moreover, the feature bits field **222** provides status information for use in virtual to physical address translations. In contrast to prior art systems, the feature bits of one embodiment of the present invention provide substantial design flexibility by enabling software to change the format of the GART table **210** (FIG. 5*a*) without the need for a costly redesign of the hardwired circuitry. For example, during an address translation, the system may need to verify that the physical address corresponding to the virtual address still includes valid data. Similarly, the system may need to determine if a referenced physical address has been read or written to. The contents of the feature bits field **222** provide this functionality. In one embodiment, the feature bits field **222** includes indicators for PTE valid **226**, page read **228** and page write **230**. These indicators **226**, **228**, **230** may be located anywhere within the feature bits field **222** and may be implemented using at least one bit. To implement an indicator, such as PTE valid **226**, the present invention uses a mask register loaded during system boot up. Thus, for PTE valid **226**, the ValidMask register is used to select the bit(s) to set in the feature bits field **222** to indicate a valid PTE. Similarly, for page read **228**, the ReadMask register is used to select the bit(s) to set when a translated address has been read. Furthermore, for a page write **230**, the WriteMask register is used to select the bit(s) to set when a translated address has been written to. For example, if ValidMask is zero, then no PTE Valid **226** indicator is defined. Otherwise, a PTE Valid **226** mask is defined and can be applied to a GART PTE **220** to determine if the physical address translation is valid. The following VHDL pseudo-code implements this logic:

```
if ((ValidMask=0) or ((ValidMask and PTE)=ValidMask))
  then
  PTEValid :=true;
else
  PTEValid :=false;
end if;
```

Similarly, to implement the page read **228** and page write **230** indicators, a logical OR operation is performed on the GART PTE **220** using the WriteMask during write operations and with the ReadMask during read operations. The resulting GART PTE **220** is then written to memory **218** (FIG. 5*a*) to provide the page read **228** or page write **230** status information. In a similar fashion, if the WriteMask or ReadMask is zero, then no page write **230** or page read **228** indicator is defined and the GART PTE **220** is not written to memory. The following VHDL pseudo-code implements the page write **230** and page read **228** indicators:

```
if ((WriteMask 0) and ((PTE and WriteMask)
  WriteMask))
then
  PTE :=PTE or WriteMask;
  UpdatePTE :=true;
else
  PTE :=PTE;
  UpdatePTE :=false;
end if;
if ((ReadMask 0) and ((PTE and ReadMask) ReadMask))
then
  PTE :=PTE or ReadMask;
  UpdatePTE :=true;
else
  PTE :=PTE;
  UpdatePTE :=false;
end if;
```

As discussed previously, the indicators **226**, **228**, **230** may be implemented by programming a mask register during system boot up. In one embodiment, the initialization BIOS programs the mask register. In another embodiment, an operating system API programs the mask register during system boot up.

For example, suppose the following mask registers:

ValidMask	:=0x00100000	—Position of Valid indicator in PTE
WriteMask	:=0x00200000	—Position of Write indicator in PTE
ReadMask	:=0x00400000	—Position of Read indicator in PTE

are programmed during system boot up. To determine if the contents of a PTE **220** are valid, the if condition:

((ValidMask=0) or ((ValidMask and PTE)=ValidMask)) is evaluated to determine if it is true or false. Referring back to the previous example, note that PTE(2)= 0x12345678. Now, substitute the values of PTE(2) and ValidMask into the if condition:

((ValidMask=0) or ((ValidMask and PTE)=ValidMask)) -or-

((0x00100000=0) or ((0x00100000 and 0x12345678)= 0x00100000)) -or-

((0x00100000=0) or (0x00100000=0x00100000))

to calculate a result. Here, the result is true indicating that the PTE is valid. Similarly, for a ValidMask set to 0x01000000, evaluation of the if condition proceeds as follows:

((ValidMask=0) or ((ValidMask and PTE)=ValidMask)) -or-

((0x01000000=0) or ((0x01000000 and 0x12345678)= 0x01000000)) -or-

((0x0100000=0) or (0x00000000=0x01000000))

to produce a false result as both (0x0100000 0) and (0x00000000 0x01000000), indicating an error reporting and recovery procedure is needed. Moreover, for a ValidMask set to 0x00000000 (i.e., valid bit disabled), the if condition always evaluates to true indicating that no errors are present.

In a similar fashion, for a write transaction, the if condition:

((WriteMask 0) and ((PTE and WriteMask) WriteMask))

is evaluated. Note that the expression (WriteMask 0) indicates that the write mask is enabled and the expression ((PTE and WriteMask) WriteMask)) determines if the write bit of PTE(2) has already been set. Now, for PTE(2)=0x12345678 and WriteMask=0x00200000, substitute these values into the if condition:

((WriteMask 0) and ((PTE and WriteMask) WriteMask)) -or-

((0x00200000 0) and ((0x12345678 and 0x00200000) 0x00200000)) -or-

((0x00200000 0) and (0x00200000 0x00200000)) to produce a false result as 0x00200000=0x00200000. Thus, the write bit does not need to be set. However, if PTE(2)=0x12145678, the if condition evaluates as follows:

((WriteMask 0) and ((PTE and WriteMask) WriteMask)) -or-

((0x00200000 0) and ((0x12145678 and 0x00200000) 0x00200000)) -or-

((0x00200000 0) and (0x00000000 0x00200000))

to produce a true result as 0x00000000 0x00200000. Here, the write bit for PTE(2) is set as the if condition indicates that a write has not occured on this page before and the contents of PTE(2) are calculated as follows:

PTE :=PTE or WriteMask; -or-

PTE(2):=0x12145678 or 0x00200000 -or-

PTE(2):=0x12345678

and PTE(2)=0x12345678 is written back to memory.

Lastly, for a read transaction, the if condition:

((ReadMask 0) and ((PTE and ReadMask) ReadMask))

is evaluated. This pseudo-code operates in a substantially similar manner to the pseudo-code discussed above for the WriteMask. Note that the expression (ReadMask 0) indicates that the read mask is enabled and the expression ((PTE and ReadMask) ReadMask)) determines if the read bit of PTE(2) has already been set. Now, for PTE(2)=0x12345678 and ReadMask=0x00400000, substitute these values into the if condition:

((ReadMask 0) and ((PTE and ReadMask) ReadMask)) -or-

((0x00400000 0) and ((0x12345678 and 0x00400000) 0x00400000)) -or-

((0x00400000 0) and (0x00000000 0x00400000)) to produce a true result as 0x00000000=0x00400000. Thus, PTE(2) has not been read before and the value of PTE(2) is calculated as follows:

PTE :=PTE or ReadMask; -or-

PTE(2):=0x12345678 or 0x00400000 -or-

PTE(2):=0x12745678

and PTE(2)=0x12745678 is written back to memory.

FIG. 6*a* illustrates the translation of a virtual address **200** to a physical address **202** (FIG. 5*a*) using a translation look aside buffer (TLB) **240**. As before, a virtual address **200** includes a virtual page number field **204** and an offset field **206**. Translation of the virtual page number field **204** occurs by finding a PTE of the GART table **210** corresponding to the contents of the virtual page number field **204**. To identify the PTE, the GART base address **212** is combined at **213** with the contents of the virtual page number field **204** to obtain a PTE address **214**. The PTE address **214** in turn provides the physical page number **216** corresponding to the virtual page number **204**. However, at this point, a TLB entry **242** is formed having a virtual page field **244**, its corresponding physical page field **246**, a least recently used (LRU) counter **248** to determine the relative age of the TLB entry **242** and a status indicator **250** to determine when the TLB **240** has valid information. The TLB entry **242** is stored in a TLB **240** having a plurality of TLB entries **252**. In one embodiment, there are a sufficient quantity of TLB entries **252** to cover all of the translatable addresses in the entire GART range **184** (FIG. 4). In this embodiment, system logic **154** (FIG. 3) includes a block of registers to implement the TLB **240**. In another embodiment, system logic **154** (FIG. 3) includes a fast memory portion, such as cache SRAM, to implement the TLB **240**.

FIG. 6*b* illustrates the use of registers to provide direct read and write access to the TLB entries **252**. In one embodiment, a TLB **240** operates as a memory cache for the most recently used PTEs. In contrast, the interface of FIG. 6*b* enables direct access of TLB entries **252** to reduce latency and memory requirements. In this embodiment, control logic **232** receives a configuration bit from the processor **152** (FIG. 3) to disable the cache like operation of the TLB **240**, thus enabling a direct access mode to the TLB **240** controlled by software. In the direct access mode, the processor **152** (FIG. 3) loads a TLB address into Address Register **234**. Control logic **232** provides the TLB address in Address Register **234** to the Mux **238** for selection of a TLB entry referenced by the TLB address. In a read operation, the TLB **240** returns the contents of the TLB entry referenced by the TLB address to the Mux **238**, which in turn passes the contents of the TLB entry to the Data Register **236** for storage. The processor **152** (FIG. **3**) then reads the Data Register **236** to obtain the contents of the desired TLB address. In a write operation, the processor **152** (FIG. 3) loads data to be written to the TLB **240** into the Data Register **236**. Control logic **232** provides the data in Data Register **236** to the Mux **238**, which then passes the data to the TLB **240** for storage in the TLB entry referenced by the TLB address stored in Address Register **234**.

For example, suppose the processor **152** (FIG. 3) needs to update TLB(1) with the value 0x12345678 and verify storage of 0x12345678 in TLB(1). The processor **152** (FIG. 3) writes the TLB address corresponding to TLB(1) into the Address Register **234** and the value 0x12345678 into Data Register **236**. Control Logic **232** provides the contents of Address Register **234**, the TLB(1) address, to the Mux **238** for selection of TLB(1). The Mux **238** then passes the value 0x12345678 from Data Register **236** to the TLB **240** for storage in TLB(1). To verify the write operation, the processor **152** (FIG. 3) now executes a read command. As Address Register **234** still holds the TLB(1) address, control logic **232** provides the TLB(1) address from Address Register **234** to the Mux **238** for selection of TLB(1). The TLB **240** returns the contents of the TLB(1), 0x12345678, to the Mux **238**, which in turn passes the value 0x12345678 to the Data Register **236** for access by the processor **152** (FIG. 3). In this manner, the embodiment of FIG. 6*b* provides a mechanism for indirect addressing, whereby individual TLB entries may be directly accessed.

FIG. **7** illustrates the operation of a TLB **240** to provide translation of a virtual address **200** to a physical address **202** to retrieve the desired information from the main memory **218**. The TLB **240** comprises a plurality of TLB entries **252**, each entry having a virtual page field as described with reference to FIG. 6*a*. To determine if a desired translation exists in the TLB **240**, the contents of the virtual page number field **204** are compared at **253** to the contents of the virtual page fields of each of the plurality of TLB entries **252** in the TLB **240**. For example, the contents of the virtual page field **246** (FIG. 6*a*) of TLB entry **242** (FIG. 6*a*) are compared at **253** to the contents of the virtual page number field **204** (FIG. 7) and no match is found. Upon finding a match, an index **254** corresponding to the matching TLB entry **255** is used to retrieve the contents of the matching TLB entry **255** from the TLB **240**. The contents of the physical page field **256** of the matching TLB entry **255** are combined at **217** with the contents of the offset field **206** of the virtual address **200** to form the physical address **202**, which references a location in main memory **218** holding the desired information. Note that a status indicator **262** of the matching TLB entry **255** indicates whether the contents of the physical page field **256** are valid and, if so, a LRU counter **260** is updated.

Referring now to FIG. **8**, a flowchart illustrates a method of using the present invention. At state **300**, the system logic **154** (FIG. 3) receives an AGP request for data referenced by a virtual address **200** (FIG. 6*a*). At state **302**, the system logic **154** (FIG. 3) determines if the TLB **240** (FIG. 6*a*) has the requested virtual address **200** (FIG. 6*a*). If the requested virtual address **200** (FIG. 6*a*) is not in the TLB **240** (FIG. 6*a*), the system logic **154** obtains the virtual to physical address translation from the GART table **210** (FIG. 6*a*) located in main memory **218** (FIG. 6*a*). At state **304**, the PTE Address **214** (FIG. 6*a*) is generated by combining the GART base address **212** (FIG. 6*a*) with the contents of the virtual page number field **204** (FIG. 6*a*) of the virtual address **200** (FIG. 6*a*). At state **306**, the system logic **154** (FIG. 3) fetches a GART PTE **220** (FIG. 5*b*) corresponding to the PTE Address **214** (FIG. 6*a*) from the main memory **218** (FIG. 6*a*). Upon retrieving the GART PTE **220** (FIG. 5*b*), the system moves to state **308** wherein a TLB entry slot **242** (FIG. 6*a*) in the TLB **240** (FIG. 6*a*) is selected to store the physical translation information for the virtual address **200** (FIG. 6*a*). The virtual to physical address translation proceeds to state **310** as for the circumstance where the requested virtual address **200** (FIG. 6*a*) exists in the TLB **240** (FIG. 6*a*).

At state **310**, the LRU counters **248** (FIG. 6*a*) of all TLB entries **252** (FIG. 6*a*) are updated to reflect the most recent access of the TLB **240** (FIG. 6*a*). At state **312**, the physical address **202** (FIG. 7) corresponding to the virtual address **200** (FIG. 7) is formed by combining the contents of the physical page field **256** (FIG. 7) with the offset **206** (FIG. 7) of the virtual address **200** (FIG. 7). At state **314**, the System logic **154** (FIG. 3) then issues a memory request to retrieve the contents of the physical address **202** (FIG. 7). Lastly, the AGP request is completed at state **316**.

Referring now to FIG. **9**, a flowchart illustrates one embodiment of a process for updating the LRU counters of all TLB entries **310** (FIG. **8**). At state **320**, the LRU counter for the selected TLB entry **242** (FIG. 6*a*) is saved for subsequent comparison to the LRU counters of each of the TLB entries. This comparison takes place at state **322**. If the current TLB entry for comparison is determined to be the same as the selected TLB entry **242** (FIG. 6*a*) at state **324**, the LRU counter of the selected TLB entry **242** (FIG. 6*a*) is set to the maximum value at state **326**. Otherwise, the LRU counter of the TLB entry for comparison is decremented at state **328**. In one embodiment, the LRU counter is decremented by one. Thus, when a TLB hit occurs, the LRU counter of the selected TLB entry **255** (FIG. 7) is loaded to its maximum value and the LRU counters of all other TLB entries **252** (FIG. 7) are decremented.

Referring now to FIG. **10**, a flowchart illustrates one embodiment of a process for selecting a TLB slot **308** (FIG. **8**). At state **340**, system logic **154** (FIG. 3) determines if a TLB slot is not currently used. If an unused TLB slot is found, this slot is selected at state **342** to store the physical translation information in the TLB entry **242** (FIG. 6*a*). Otherwise, the LRU counters of all TLB slots are compared at state **344**. When the TLB entry having the minimum LRU counter value is found, this slot is selected at state **346** to store the physical translation information in the TLB entry **242** (FIG. 6*a*). Lastly, at state **348**, the status indicator **250** (FIG. 6*a*) of the selected TLB slot is set to indicate a valid entry.

Referring now to FIG. **11**, a flowchart illustrates one embodiment of a process for fetching a GART PTE **306** (FIG. **8**). At state **360**, the system logic **154** (FIG. 3) obtains the virtual page number **204** (FIG. 5*a*) from the virtual address **200** (FIG. 5*a*). At state **362**, the virtual page number **204** (FIG. 5*a*) is then combined with the GART base address **212** (FIG. 5*a*) to form a PTE Address **214** (FIG. 5*a*). Lastly, at state **364**, system logic **154** (FIG. 3) reads the PTE from main memory **218** (FIG. 5*a*) using the PTE Address **214** (FIG. 5*a*).

The present invention advantageously overcomes several limitations of existing technologies and alternatives. For example, current technologies store graphics data in expensive local frame buffer memory. In contrast, the present invention enables storing, addressing and retrieving graphics data from relatively inexpensive main memory without the bandwidth limitations of current system bus designs. Furthermore, by defining the GART in software, the present invention eliminates many hardware dependencies. Instead of expensive circuit redesigns and fabrication, the present invention enables inexpensive software modifications to address future partitioning and remapping circuitry as well as any current or future compatibility issues. Moreover, the present invention enables computer manufacturers to investigate cost and performance compromises at the system integration stage rather than at the hardware design and development stage. For example, computer manufacturers may implement the entire GART in main memory (instead of registers) to reduce register costs, while caching an image of the most recently used GART entries in a few registers to reduce access times to main memory. The invention thus provides substantial flexibility to address ever changing cost and performance requirements well after the completion of the hardware design. In contrast to existing hardware design paradigms, the present invention enables rapid and inexpensive modifications to address evolving customer and market needs.

In addition, the present invention is useful for computer system applications that flexibly allocate memory resources which are tightly coupled to the computer hardware. For example, the invention is useful in situations where hardware ascertains and reports state information, such as diagnostic data or vital product data. The invention allows for flexible reporting of the state information under software control, instead of hardware control where functions are hardwired into circuitry. Similarly, the invention provides alternate mechanisms to access internal registers for diagnostic purposes. Lastly, the invention provides a mechanism whereby status can be flexibly programmed into memory. Thus, the invention enables any device, such as a network device broadcasting bits in a serial stream, to flexibly indicate status information using a medium other than memory.

The invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiment is to be considered in all respects only as illustrative and not restrictive and the scope of the invention is, therefore, indicated by the appended claims rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced with their scope.

What is claimed is:

1. An apparatus for graphic address remapping of a virtual address, comprising:

a processor;

an interface that is accessible by the processor; and

a translation lookaside buffer (TLB) in communication with the interface, the TLB having at least one TLB entry including information which is used to translate the virtual address to a physical address;

wherein the interface receives a portion of the virtual address and provides access to the TLB entry corresponding to the portion of the virtual address, wherein the TLB entry includes translation information from a graphics address remapping table that contains location information of a plurality of physical pages of memory that are used to store graphics data, wherein the processor is capable of modifying the contents of the TLB via the interface.

2. The apparatus of claim 1, wherein the interface provides read access to the TLB entry.

3. The apparatus of claim 1, wherein the interface provides write access to the TLB entry.

4. The apparatus of claim 1, wherein the interface further comprises:

a data register;

an address register receiving a portion of the virtual address; and

a multiplexer in communication with the address register, the TLB and the data register, wherein the multiplexer selects the TLB entry corresponding to the portion of the virtual address and provides access to the selected TLB entry using the data register.

5. The apparatus of claim 1, wherein the portion of the virtual address comprises a virtual page number field.

6. The apparatus of claim 1, wherein the at least one TLB entry further comprises a least recently used (LRU) counter.

7. The apparatus of claim 1, wherein the at least one TLB entry further comprises a status indicator to indicate if the TLB entry is valid.

8. The apparatus of claim 1, wherein the virtual address includes a virtual page number field and an offset field.

9. An apparatus for graphic address remapping of a virtual address, comprising:

a processor;

an interface that is accessible by the processor; and

a translation lookaside buffer (TLB) in communication with the interface, the TLB having at least one TLB entry including information which is used to translate the virtual address to a physical address;

wherein the interface receives a portion of the virtual address and provide access to the TLB entry corresponding to the portion of the virtual address, wherein the TLB entry includes translation information from a graphics address remapping table that contains location information of a plurality of physical pages that are used to store graphics data, wherein the TLB includes at least one TLB entry for each physical page of memory that is managed by the graphics address remapping table.

10. The apparatus of claim 9, wherein the interface provides read access to the TLB entry.

11. The apparatus of claim 9, wherein the interface provides write access to the TLB entry.

12. The apparatus of claim 9, wherein the interface further comprises:

a data register;

an address register receiving a portion of the virtual address; and

a multiplexer in communication with the address register, the TLB and the data register, wherein the multiplexer selects the TLB entry corresponding to the portion of the virtual address and provides access to the selected TLB entry using the data register.

13. The apparatus of claim 9, wherein the portion of the virtual address comprises a virtual page number field.

14. The apparatus of claim 9, wherein the at least one TLB entry further comprises a least recently used (LRU) counter.

15. The apparatus of claim 9, wherein the at least one TLB entry further comprises a status indicator to indicate if the TLB entry is valid.

16. The apparatus of claim 9, wherein the virtual address includes a virtual page number field and an offset field.

17. An apparatus for graphic address remapping of a virtual address, comprising:

a processor;

an interface that is accessible by the processor; and

a translation lookaside buffer (TLB) in communication with the interface, the TLB having at least one TLB entry including information which is used to translate the virtual address to a physical address;

wherein the interface receives a portion of the virtual address and provides access to the TLB entry corresponding to the portion of the virtual address, wherein the TLB entry includes translation information from a graphics address remapping table that contains location information of a plurality of physical pages that are used to store graphics data, wherein size of the graphics address remapping table is configurable by a program that is executing on the processor.

18. An apparatus for graphic address remapping of a virtual address, comprising:

a processor;

an interface that is accessible by the processor; and

a translation lookaside buffer (TLB) in communication with the interface, the TLB having at least one TLB entry including information which is used to translate the virtual address to a physical address;

wherein the interface receives a portion of the virtual address and provides access to the TLB entry corresponding to the portion of the virtual address, wherein the TLB entry includes translation information from a graphics address remapping table that contains location information of a plurality of physical pages that are used to store graphics data, wherein size of the graphics address remapping table is configurable by a program that is executing on the processor, and wherein TLB includes at least one TLB entry for each physical page of memory that is managed by the graphics address remapping table.

* * * * *

US005708790A

United States Patent [19]

White et al.

[11] **Patent Number: 5,708,790**

[45] **Date of Patent: Jan. 13, 1998**

[54] **VIRTUAL MEMORY MAPPING METHOD AND SYSTEM FOR ADDRESS TRANSLATION MAPPING OF LOGICAL MEMORY PARTITIONS FOR BAT AND TLB ENTRIES IN A DATA PROCESSING SYSTEM**

[75] Inventors: **Steven W. White; G. Jeanette McWilliams**, both of Austin; **Jack Wayne Kemp**, Round Rock, all of Tex.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **570,957**

[22] Filed: **Dec. 12, 1995**

[51] Int. Cl.^{6} **G06F 12/00**; G06F 12/02; G06F 9/26; G06F 9/32

[52] U.S. Cl. **395/413**; 395/417; 395/418; 395/419; 395/421.1

[58] **Field of Search** 395/413, 416, 395/417, 418, 419, 410, 421.1, 412

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,754,394	6/1988	Brantley, Jr. et al.	395/405
4,855,900	8/1989	Simpson et al.	395/850
5,058,003	10/1991	White	395/419
5,109,485	4/1992	Smith, Jr.	395/200.08
5,117,350	5/1992	Parrish et al.	395/401
5,210,844	5/1993	Shimura et al.	395/480
5,442,766	8/1995	Chu et al.	395/414
5,450,558	9/1995	Ludwig	395/418
5,535,351	7/1996	Peng	395/417

Primary Examiner—Robert W. Downs
Assistant Examiner—Than V. Nguyen
Attorney, Agent, or Firm—Richard A. Henkler; Brian F. Russell; Andrew J. Dillon

[57] **ABSTRACT**

A method and system for address translation mapping of logical partitions for address translation buffer entries in a data processing system is provided. The method comprises receiving a logical address for a memory reference to a selected logical partition of a plurality of logical partitions of a particular block of virtual memory, wherein the block of virtual memory is divided into the plurality of logical partitions, and wherein the logical address includes a plurality of logical partition selection bits selecting the selected logical partition from among the plurality of logical partitions. If the selected logical partition is valid in real memory, as indicated by a logical partition valid bit associated with the selected logical partition, a physical address for the memory reference in the selected logical partition is compiled from an entry of an address translation buffer that is associated with the particular block of virtual memory, wherein the logical partition valid bit is one of a plurality of logical partitions valid bits contained in the entry associated with the particular block of virtual memory, the plurality of logical partition valid bits being associated with the plurality of logical partitions. Thereafter, the memory reference within the selected logical partition is retrieved at the compiled physical address.

22 Claims, 2 Drawing Sheets

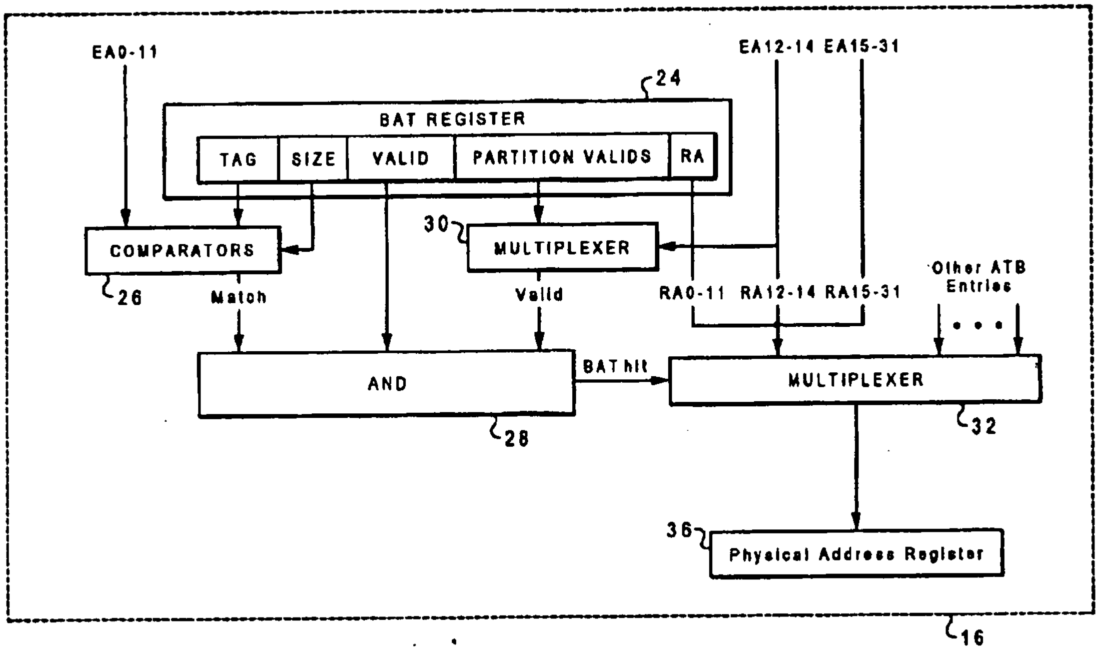
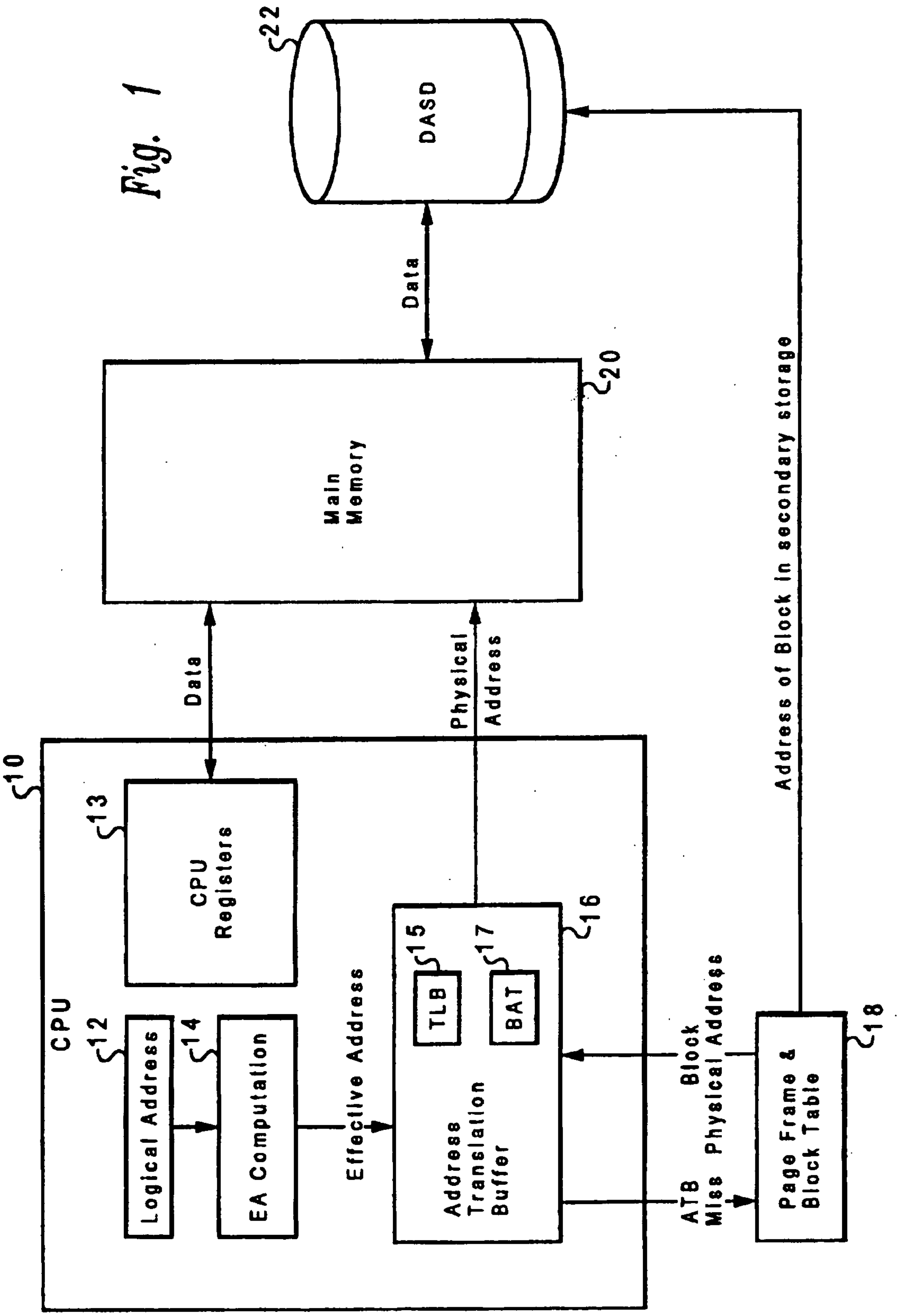


Fig. 1



22
DASD
Data
20
Main Memory
Data
Physical Address
Address of Block in secondary storage
10
CPU
13
CPU Registers
12
Logical Address
14
EA Computation
Effective Address
16
15
TLB
17
BAT
Address Translation Buffer
Block Physical Address
ATB Miss
18
Page Frame & Block Table

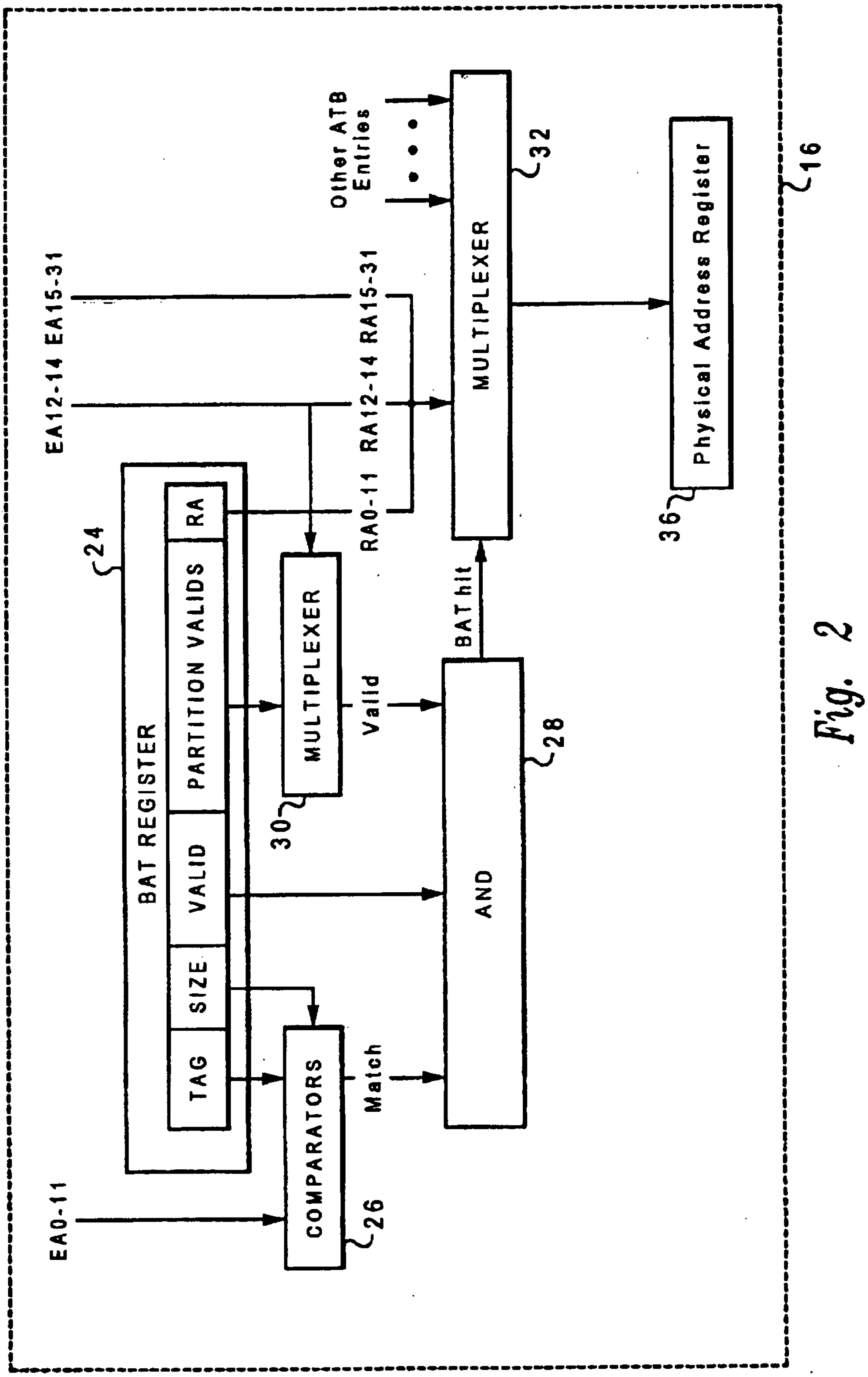

EA0-11
EA12-14
EA15-31
RA0-11
RA12-14
RA15-31
Other ATB Entries
MULTIPLEXER
32
Physical Address Register
36
16
BAT REGISTER
TAG
SIZE
VALID
PARTITION VALIDS
RA
24
MULTIPLEXER
30
Valid
BAT hit
AND
28
COMPARATORS
26
Match

Fig. 2

VIRTUAL MEMORY MAPPING METHOD AND SYSTEM FOR ADDRESS TRANSLATION MAPPING OF LOGICAL MEMORY PARTITIONS FOR BAT AND TLB ENTRIES IN A DATA PROCESSING SYSTEM

CROSS-REFERENCE TO RELATED APPLICATION

The present application is related to U.S. Pat. Application Ser. No. 08/571,066, entitled "Virtual Memory Mapping Method and System For Memory Management Of Pools Of Logical Partitions For BAT And TLB Entries In A Data Processing System", filed herewith by the Inventors hereof and assigned to the assignee herein.

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to a virtual memory mapping system in a data processing system, and in particular to an improved virtual memory mapping system in a data processing system having cached address translation mapping of memory references. Still more particularly, the present invention relates to an improved virtual memory mapping system in a data processing system having fine granularity of cached address translation mapping of memory blocks.

2. Description of the Related Art

A computer system typically includes a processor coupled to a hierarchically staged storage system. The computer's hierarchy of storage devices comprises primary memory that includes internal components such as the CPU registers, cache memory, and main memory, and secondary memory that includes any external storage devices such as disks or tapes. Main memory is typically a DRAM or a SRAM. Computers often use an intermediate high-speed buffer called a cache memory that resides between the external devices and main memory or between main memory and the CPU. Cache memories speed up the apparent access times of the slower memories by holding the words that the CPU is most likely to access. The hardware can dynamically allocate parts of the memory within the hierarchy for addresses deemed most likely to be accessed soon.

Most computers use a multilevel storage system that operates as a virtual memory. In such systems, most programs are stored on an external device, such as a hard disk. In practice, the logical-address space of many computers is much larger that their physical-address space in main memory. For example, if a byte-addressed computer uses a 32-bit address, its logical address space has 2^{32} memory locations, which is four gigabytes (GB). The operating system loads the program into the main memory in parts or pages, as demanded for execution. By using virtual memory paging, the computer loads into main memory only those parts of a program that it currently needs for execution. The remainder of the program resides in external storage until needed. Thus, one of the biggest advantages of virtual memory is that because programs are stored on secondary storage devices, the size of a program that may be executed is limited not by the size of main memory but rather by the size of the computer's logical-address space.

Because various blocks of memory may be stored throughout the memory hierarchy, a program's logical addresses to particular instructions or data may no longer correspond to the physical addresses for the particular block of memory containing those instructions or data. In a virtual memory system, the operating system maintains special tables that keep track of where each part of the program resides in main memory and in external storage. The memory map between logical-address space and physical-address space is maintained in a page frame or block table having a plurality of entries, each table entry holding information about a specific page or block of memory. Thus, the CPU uses address translation mapping from the tables to translate the program's effective (or logical) addresses into their corresponding physical addresses. Most virtual memory systems keep these translation tables in main memory, and maintain a translation table base register that points to the translation table in memory. Depending upon the configuration, separate tables may exist for the block table and the page frame table.

Virtual memory hardware divides logical addresses into two parts—the virtual block (page) number (the high-order bits), and the word offset (the low-order bits). The virtual block (page) number serves as an offset into the block (or page frame) table. Therefore, when the system loads a block (or page) into memory, it always places the block beginning at a block boundary. A typical table entry includes a validity bit, which indicates whether the block is in main memory, a dirty bit, which indicates whether the program has modified the block, protection bits, which indicate which users may access the page or block of memory and how, and the page-frame or real block number (i.e., the physical address) for the block of memory, if the block (or page) is in main memory.

To minimize the amount of time required to translate a virtual address to a real address, virtual memory mapping generally uses address translation buffers to cache information for recently translated pages. Each entry of an address translation buffer holds a real block number and the same information contained in the translation table, including the validity bit for the block, one or more dirty bits, protection bits, and the virtual block number to provide the map of the virtual block number to the real block number. In paging virtual memory systems, this address translation buffer is called a translation lookaside buffer (TLB). In addition, some virtual memory systems provide an additional address translation buffer called a block address translation (BAT) buffer, which includes an additional field indicating the size of the block of memory mapped by the entry to support variable sized blocks.

In operation, whenever the CPU generates an effective address, it is sent to the TLB and the BAT, which produce the real page frame or BAT block number, if either buffer holds an entry for the referenced block of memory. If one of the address translation buffers has an entry for the reference, the effective address is translated into the physical address by concatenating the real block number held in the entry with the word offset of the effective address. If the TLB or BAT has no entry for the referenced block of memory, the hardware (or software) consults the translation table in main memory by using the virtual block number as an offset into the translation table. If the validity bit for the entry in the translation table indicates the block is in memory, the hardware copies the translation table entry and uses the real block number to access the memory into the TLB. Otherwise, the hardware initiates a trap called a page or BAT block fault, at which point the operating system intervenes to load the demanded block of memory into main memory and updates the translation table and address translation buffers. A block or page fault is an exception that instructs the operating system to load into main memory the requested or demanded block or page and to update the memory map.

In many systems, the page size is 4 kilobytes (4 KB) and there are 256 or fewer TLB entries. Consequently, the maximum amount of real storage covered by cached page translation information is often 1 MB or less. Consequently, it is unlikely that TLBs for 4 KB pages will ever cover significant portions of large (512 MB-4 GB) main memories. These TLBs are even inadequate to prevent significant performance degradation due to TLB misses while accessing data which fits in a large (4 MB–16 MB) Level Two (L2) cache. As real memory capacities, program footprints, and user working sets continue to grow, it is beneficial to increase the amount of real memory covered by cached translation information. Three common approaches to increasing the coverage are 1) increasing the number of TLB entries, 2) supporting larger pages, and 3) adding BAT facilities to augment the TLBs.

Increasing the number of TLB entries becomes expensive, both in terms of chip area and time to search for a match. Increasing the page size to 1 MB is one way to allow a limited number (64–256) of TLB entries to cover not only the L2 cache but large portions of main memory; however, the larger granularity of storage blocks and memory mapping creates great inefficiencies.

BAT facilities are an alternative, and more common way of providing many of the coverage benefits of large pages. BAT registers, which are set by the operating system, in contrast to TLB entries, which are typically reloaded by hardware, specify the translation for a block of storage which is large relative to a 4 KB page. Unlike TLB entries which typically translate a fixed-size block (a page) of storage, bits in the BAT registers allow a specification of a range roughly equivalent to 50–5000 (4 KB) pages. Both large pages and BAT blocks must be on a boundary similar to the size of the page or block in both address spaces (i.e., a 2 MB block must be on a 2 MB boundary). While large pages or large BAT blocks can alleviate the pressure for more TLB entries, the larger blocks of memory create operating system complexities and granularity issues that create great disadvantages for the memory mapping system. The major disadvantages of such large blocks are:

- When a BAT fault (i.e., a "page fault" for a portion of memory which will be covered by a BAT entry) is encountered, the faulting process is suspended until all data for the large block has been brought from disk to memory and the translation is made valid for the block. Longer waits are associated with larger blocks.
- While an entry is valid, all data must be present. Even if only a small portion of the data (or instructions) is needed, the entire block of main storage is allocated (and unavailable for other uses). Hardware complexity of BATs results in implementations with only a few BAT entries. In multi-user .systems with hundreds or thousands of processes and where memory is a limited resource, operating systems are reluctant to allocate large contiguous blocks of real memory to each user process. Therefore, it is expected that operating systems will not generally allow BAT entries (as currently defined) to be used for user data/instructions.
- It is difficult to provide coverage for areas which are not a power-of-two bytes because blocks must be stored on a power-of-two boundary. For example, to cover a 7 MB block, there are two choices: 1) Use an 8 MB entry and waste 1 MB of real storage. (This may not be permissible since the user can access storage beyond the expected 7 MB limit.). 2) Use three BAT entries (4 MB+2 MB+1 MB). This is a serious burden for current BAT hardware as most implementations have only a few (2–8) BAT entries.

- Variable sized blocks, as are used in BATs, require more advanced memory management techniques from the operating system than do uniform sized pages. When a BAT fault occurs, the operating system must find (or create) an available block of contiguous real storage for the entire block, a process that may require the removal of some blocks already in memory.
- Placing differing sized blocks in memory can leave fragments or splinters, i.e., small blocks of memory between other allocated blocks that are too small to be used by other blocks, creating inefficiency.

It can be seen that there is a need for a virtual memory mapping system that provides translation coverage for large blocks of storage, while still providing sufficiently fine granularity to improve data input/output (I/O) efficiency and reduce memory allocation problems. Such a virtual memory mapping system would dramatically increase the coverage of cached translation information while avoiding the problems associated with coarser granularity.

SUMMARY OF THE INVENTION

According to the present invention, a method and system for address translation mapping of logical partitions for address translation buffer entries in a data processing system is provided. The method comprises receiving a logical address for a memory reference to a selected logical partition of a plurality of logical partitions of a particular block of virtual memory, wherein the block of virtual memory is divided into the plurality of logical partitions, and wherein the logical address includes a plurality of logical partition selection bits selecting the selected logical partition from among the plurality of logical partitions. If the selected logical partition is valid in real memory, as indicated by a logical partition valid bit associated with the selected logical partition, a physical address for the memory reference in the selected logical partition is compiled from an entry of an address translation buffer that is associated with the particular block of virtual memory, wherein the logical partition valid bit is one of a plurality of logical partitions valid bits contained in the entry associated with the particular block of virtual memory, the plurality of logical partition valid bits being associated with the plurality of logical partitions. Thereafter, the memory reference within the selected logical partition is retrieved at the compiled physical address.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. However, the invention, as well as a preferred mode of use, and further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 depicts the virtual memory mapping system of a preferred embodiment of the present invention; and

FIG. 2 shows a block diagram of an Address Translation Buffer, in accordance with a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

With reference now to the figures, and in particular with reference to FIG. 1, there is depicted the virtual memory mapping system of a preferred embodiment of the present invention. CPU 10 executes programs stored in the logical address space of direct access storage device (DASD) 22.

When an instruction executed in CPU 10 generates a memory reference to an instruction (or data) not contained within the CPU registers 13, the memory reference must be retrieved from the data processing system's memory hierarchy.

The instruction's logical address 12 addresses a block of memory within the logical address space of the virtual memory system. The block of memory addressed by the logical address is partitioned into a plurality of logical partitions. An effective address is computed from the logical address by the effective-address computation hardware 14. During program execution, the effective-address computation hardware 14 converts the instruction's address specifications into effective addresses. These are the addresses that the CPU uses when referencing an instruction or variable, and in general they agree with the compiler's logical address (as will be assumed herein).

CPU 10 sends the effective address to Address Translation Buffer (ATB) 16. Address Translation Buffer 16 contains TLB 15 and BAT 17. Both the TLB and the BAT are searched for an entry for that effective address. According to the present invention, a memory reference for a particular instruction may be to a selected logical partition of the block of virtual memory addressed by the logical address. A logical address includes a virtual block number addressing the block of memory in virtual memory space, a plurality of logical partition selection bits selecting a partition of the block of virtual memory, and a real address offset. In the present embodiment, only block entries of BAT 17 may be logically partitioned, but it will be understood by those skilled in the art that the present invention is applicable to any ATB entry, such as large page TLB entries. If Address Translation Buffer 16 has a valid entry for that effective address, it generates the corresponding physical address for the memory reference, which indicates the location of the memory reference within main memory 20. The physical address is transferred from Address Translation Buffer 16 to main memory 20, thereby accessing the memory reference in the physical address space and loading it from main memory 20 to CPU registers 13.

If Address Translation Buffer 16 has no entry for the effective address, the hardware (or software) consults the block (or page frame) table 18, using the virtual block number as an offset into the block table. (Although shown separately, block table 18 typically resides within main memory 20.) Block table 18 contains many more entries than Address Translation Buffer 16, but provides a much slower access time because it is contained in the slower main memory 20 and has a larger number of entries to search when making a comparison with the effective address. If the validity bit for the entry in the block table 18 corresponding to the virtual block number indicates the block is in memory, the hardware copies the block table entry into a new entry in the Address Translation Buffer 16 and uses the real (physical) block number in the block table entry to access the addressed data in main memory 20. Otherwise, the search of the block table for a memory reference to a block of memory that is not present in main memory 20 results in a block (or page) fault. A block fault is an exception that instructs the operating system to load into main memory the requested or demanded block and to update the block map (i.e., create an entry in the block table 18 and address translation buffer 16). The retrieved data is then loaded from main memory 20 to CPU 10 to satisfy the memory access.

As will be appreciated, the memory hierarchy shown in FIG. 1 may also include a cache memory between main memory 20 and CPU 10 that provides a high-speed RAM with a faster access time to hold the memory references most recently used. As will be appreciated, Address Translation Buffer 16 would make a request for the translated memory access to the cache memory, so that if the cache holds a copy of the requested data, the cache will quickly process the request. Otherwise, the cache forwards the request to main memory 20.

Referring now to FIG. 2, there is shown a block diagram of Address Translation Buffer 16, in accordance with a preferred embodiment of the present invention. Address Translation Buffer 16 has eight BAT registers. Each BAT register contains a BAT entry. As will be appreciated by those skilled in the art, the present invention is also applicable to large page TLB entries having partitioned pages and the operation of such an embodiment would be substantially similar to the present embodiment, except for the variable sized blocks for the BAT entries, and so such a description is not repeated herein. FIG. 2 shows a single ATB register 24 and associated hardware 26–36. TLB 15 and the remaining seven ATB registers and associated hardware of Address Translation Buffer 16 are not shown. As will be appreciated, Address Translation Buffer 16 can be equipped to provide address translation caching for any number of entries, including eight, for a preferred embodiment.

As shown in FIG. 2, each BAT entry of Address Translation Buffer 16 contains a block Valid bit, block Size, effective address (TAG), real address (RA), and a plurality of partition valid bits (Partition Valids). The block Valid bit specifies whether the contents of the BAT register represent a valid address translation for the referenced block of memory. If the block of memory referenced by the TAG in the entry is stored in main memory at the real address (RA), the block valid bit is set. The block Size is an encoding of the size, in bytes, of the block of memory translated by the current entry. While the implemented TAG and RA fields need a sufficient amount of bits to handle the smallest allowable block, the number of bits used for a given translation is determined by the block size. For a block size of 2^N bytes, the TAG field contains the address of the first byte of the block in the effective/virtual address space (i.e., the virtual block number) for the translated block of memory, with the low-order N address bits removed. The RA field contains the address of the first byte of the block in the real address space of the main memory (i.e., the real block number), with the low-order N address bits removed. Although the Address Translation Buffer supports multiple-size blocks, in the preferred embodiment, a 1 MB BAT block size is assumed.

In a preferred embodiment, the ATB entry has eight partition valid bits, each corresponding to one of eight separate partitions of the 1 MB block of virtual memory translated by the particular ATB entry. Each partition bit is set when its corresponding partition of the block of memory is stored in the main memory. In accordance with the present invention, a memory reference to a particular partition of a block of virtual memory will only result in the particular partition of the block of memory being stored to main memory. If a valid partition is stored in main memory, its corresponding partition valid bit in the partition Valids field of the ATB entry is set. If that partition becomes invalid or is overwritten, the corresponding partition valid bit is reset.

As seen in FIG. 2, an untranslated effective address EA0-31 is presented to ATB 16. This untranslated effective address is simultaneously presented to all (appropriate) ATB entries in the ATB 16. In this embodiment, the effective and real addresses for a block of memory are 32-bit addresses. The upper twelve bits (EA 0–11), which specify a 1 MB block, of the effective address are compared by comparators 26 to the upper effective address bits (TAG 0–11) for the block corresponding to this ATB entry. If these addresses match, an indication is output to AND gate 28, which forms the last stage of the tag-comparison function. The low-order effective address bits (FA 12–31 in the case of the 1 MB block) pass to multiplexer 32 unmodified to become real address bits RA 12–31.

Simultaneously, a valid bit from the eight partition valid bits (Partition Valids) is selected to determine if the block of memory referenced by the effective address is valid in the main memory. The high-order effective address bits within the block (i.e., the high-order bits addressing a particular partition within the block of memory corresponding to the ATB entry) are used to select the partition and its valid bit within the partition valid bits. Within the 1 MB block addressed by the first twelve effective address bits, the high-order effective address bits EA 12–14 select a particular partition of the eight partitions of the block. The eight partition valid bits are input into multiplexer 30 and the high-order effective address bits EA 12–14 are used to index the selected partition valid bit as an output from multiplexer 30. The output of multiplexer 30 is input into ANDgate 28 as another input into the last stage of the tag comparison function, as will be understood by those skilled in the art.

If there has been a match of the virtual block number for the received logical address and the virtual block number stored in register 24, and further, if the logical partition selection bits in the received logical address indicate a valid partition in main memory, as indicated by the selected partition bit in the ATB entry, the output of ANDgate 28 is set, indicating an ATB hit. All ATB registers are connected to multiplexer 32. The ATB hit for this ATB register selects the two address inputs (RA12–14 and 15–31) and the RA field (RA 0–11) for this entry in multiplexer 32. These selected address bits are output to physical address register 36, which stores the real block number (RA 0–11), the logical partition selection bits (RA 12–14), and the real address offset (RA 15–31). The contents of physical address register 36 are concatenated to form the physical address that is transferred to the main memory to access the selected partition of the memory reference. If no BAT or TLB entry produces an ATB hit signal, an ATB fault occurs.

As will be appreciated by those skilled in the art, the logical partition selections bits within the effective address will vary depending upon the number of partitions for the block of memory and the block size. Further, the operating system will set the fields of the ATB entry when loaded as required by the block size. Thus, the appropriate high-order effective address bits, as a function of the block size, are input into comparators 26 upon the receipt of a memory reference. Also, the appropriate logical partition selection bits of a received effective address, as a function of the block size, are used to provide the appropriate control of multiplexer 30. In addition, the selection of the referenced partition's valid bit by multiplexer 30 is an operation that can be overlapped with the existing and relatively lengthy TAG comparison. The selection of a partition, as opposed to a block, requires no additional time to generate the RA fields and the ATB hit signal, since the Andgate 28 can be implemented as one additional bit in the comparators 26.

As can be seen, the present invention provides a method and system for refining the granularity of a large block of memory by introducing logical partitions of the block and associating a valid bit with each partition. As memory references are satisfied, the system validates partitions, rather than an entire ATB entry. If a new ATB entry must be created within Address Translation Buffer 16 to satisfy a fault (i.e., the initial fault for the block), the partition will be backed by real storage in the main memory and an ATB entry is created with only the one valid bit corresponding to the particular selected partition of the block set. If the corresponding ATB entry already existed when the fault occurs (i.e., subsequent faults for the block), the faulting partition would simply be backed up by real storage in the main memory and the partition's valid bit would be set in the ATB entry. For additional performance, the operating system may chose to fetch-ahead additional partitions of the block simultaneously with, or after the immediate memory reference.

As will be appreciated by those skilled in the art, significant advantages are created by the present invention's method and system of introducing logical partitions for a ATB entry. First, the logical partitioning of the address translation mapping can be provided in either existing BAT or TLB technologies. Additionally, the response time of I/O for a fault is decreased because only a subset, a single partition, of the block is required. The faulting process may thus resume after a portion of the data (or instructions) has been retrieved, rather than waiting for the entire block covered by the ATB entry. Further, the amount of contiguous real storage required to immediately satisfy a ATB fault is reduced to the size of the partition. Additionally, the real storage associated with non-valid partitions can be made available to other processes until needed by the process. If the other portions of the block are not accessed during some interval, additional savings may result by eliminating unnecessary I/O. Still further, because the granularity of real memory which must be reserved for a particular memory access is reduced, address translation caching of user address space is more acceptable to the system. Last, the present invention much more efficiently accommodates memory references to blocks of memory which are not a power-of-two in size (such as 3 MB or 5 MB). Because the present invention requires that only selected partitions be stored at a real address in main memory, the entire non-power-of-two block is not required to be saved in main memory. Therefore, only a single ATB entry is required for this particular memory reference, and further, an entire power-of-two block of main memory does not have to be reserved for this ATB entry.

Although the present invention has been described in terms of 32-bit address spaces, it is equally applicable to other sizes, such as 64-bit addresses. Moreover, although this invention has been described in terms of eight partitions, it is equally applicable to other sizes, such as two or four partitions. Moreover, the present invention is equally applicable to TLB or BAT facilities. As will be appreciated, the operating system would need to support the logical partitioning as used in the present invention. Exploiting the features and advantages of the present invention requires that the operating system perform the manipulations of the logical partition selection bits in the ATB entry and allow for additional ATB faults for partitions that are not yet valid.

While the invention has been particularly shown and described with reference to a preferred embodiment, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention.

What is claimed is:

1. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system, the method comprising the steps of:

receiving a logical address for a memory reference to a selected logical partition among a plurality of logical partitions of a particular block of virtual memory, wherein the block of virtual memory is divided into the plurality of logical partitions, and wherein the logical address includes a plurality of logical partition selection bits selecting the selected logical partition from among the plurality of logical partitions;

if the selected logical partition is valid in real memory, as indicated by a logical partition valid bit associated with the selected logical partition, compiling a physical address for the memory reference in the selected logical partition from an entry of an address translation buffer that is associated with the particular block of virtual memory regardless of whether logical partitions of said particular block other than said selected logical partition are valid within said real memory, wherein the logical partition valid bit is one of a plurality of logical partitions valid bits contained in the entry associated with the particular block of virtual memory, the plurality of logical partition valid bits each being associated with a respective one of the plurality of logical partitions; and

retrieving the memory reference within the selected logical partition at the compiled physical address.

2. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 1, wherein a logical address includes a virtual block number for the particular block of virtual memory, and wherein each entry includes a virtual block number for an associated block of memory and a real block number addressing the associated block in real memory, and wherein an entry is associated with a particular block of virtual memory when the virtual block number for the particular block of virtual memory and the virtual block number for the entry are equal.

3. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 1, wherein the step of compiling comprises concatenating the real block number with the logical partition selection bits and a real address offset to form a physical address.

4. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 1, further comprising the steps of:

if said plurality of logical partition valid bits indicate that none of said plurality of logical partitions of said particular block of virtual memory is valid within said real memory, storing into the real memory the selected logical partition for the received logical address, and creating an entry in the address translation buffer for the particular block of memory addressed by the received logical address.

5. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 1, further comprising the steps of:

if said plurality of logical partition valid bits indicate that at least one logical partition of said particular block other than said selected logical partition is valid within the real memory, storing into the real memory the selected logical partition addressed by the received logical address, and updating the plurality of logical partition valid bits in the entry corresponding to the particular block of memory to indicate that the selected logical partition is contained in the real memory.

6. A method for address translation mapping of logical partitions for address translation buffer entries in a data

processing system according to claim 4, and further comprising the step of accessing said selected logical partition in the real memory prior to storing, into said real memory, another of said plurality of logical partitions in the block of memory addressed by the received logical address.

7. A system for address translation mapping of logical partitions for address translation buffer entries in a data processing system, said system comprising:

means for receiving a logical address for a memory reference to a selected logical partition among a plurality of logical partitions of a particular block of virtual memory, wherein the block of virtual memory is divided into the plurality of logical partitions, and wherein the logical address includes a plurality of logical partition selection bits selecting the selected logical partition from among the plurality of logical partitions;

means, responsive to the selected logical partition being valid in real memory, as indicated by a logical partition valid bit associated with the selected logical partition, for compiling a physical address for the memory reference in the selected logical partition from an entry of an address translation buffer that is associated with the particular block of virtual memory regardless of whether logical partitions of said particular block other than said selected logical partition are valid within said real memory, wherein the logical partition valid bit is one of a plurality of logical partitions valid bits contained in the entry associated with the particular block of virtual memory that are each associated with a respective one of the plurality of logical partitions; and

means for retrieving the memory reference within the selected logical partition at the compiled physical address.

8. A system for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 7, wherein a logical address includes a virtual block number for the particular block of virtual memory, and wherein each entry includes a virtual block number for an associated block of memory and a real block number addressing the associated block in real memory, and wherein an entry is associated with a particular block of virtual memory when the virtual block number for the particular block of virtual memory and the virtual block number for the entry are equal.

9. A system for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 7, wherein the means for compiling comprises means for concatenating the real 4 block number with the logical partition selection bits and a real address offset to form a physical address.

10. A system for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 7, further comprising means, responsive to said plurality of logical partition valid bits indicating that none of said plurality of logical partition of said particular block is valid within the real memory, for storing into the real memory the selected logical partition for the received logical address, and creating an entry in the address translation buffer for the block of memory addressed by the received logical address.

11. A system for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 7, further comprising means, responsive to said .plurality of logical partition valid bits indicating that at least one logical partition of said particular block other than said selected logical partition is

valid within the real memory, for storing into the real memory the selected logical partition addressed by the received logical address, and updating the plurality of logical partition valid bits in the entry corresponding to the block of memory addressed by the received logical address to indicate that the selected logical partition is contained in the real memory.

12. A system for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 7, said system further comprising means for accessing said selected logical partition in the real memory prior to storing, into said real memory, another of said plurality of logical partitions in the block of memory addressed by the received logical address.

13. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system, the method comprising the steps of:

receiving a logical address for a memory reference to a selected logical partition of a particular block of virtual memory, wherein the particular block of virtual memory is divided into a plurality of logical partitions, and wherein a logical address includes a virtual block number addressing the block of memory in virtual memory space, a plurality of logical partition selection bits selecting the selected logical partition within the block of memory, and a real address offset;

determining if there is a match between the virtual block number of the received logical address and a virtual block number for any entry in an address translation buffer, wherein each entry in the address translation buffer is associated with a block of virtual memory and includes a virtual block number, a real block number addressing the block of memory in real memory, and a plurality of logical partition valid bits, wherein a logical partition of the block of virtual memory is indicated as valid in real memory by an associated logical partition valid bit [of]among the plurality of logical partition valid bits; and

if there is a match of the virtual block numbers for the received logical address and an entry in the address translation buffer and if the selected logical partition is indicated as valid by the associated logical partition valid bit, combining the real block number with the logical partition selection bits and the real address offset to form a physical address in real memory for the memory reference in the selected logical partition, regardless of whether logical partitions of said particular block other than said selected logical partition are valid within said real memory.

14. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 13, further comprising the step of retrieving the memory reference in the selected logical partition addressed by the formed physical address.

15. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 13, if there is not a match of the virtual block number of the received logical address and the virtual block number of any entry in the address translation buffer, further comprising the steps of storing into the real memory the selected logical partition for the received logical address, and creating an entry in the address translation buffer for the block of memory addressed by the received logical address with the associated partition valid bit being set.

16. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 13, when the selected logical partition is not indicated as valid in an entry having a match with the virtual block number of the received logical address, further comprising the steps of storing into the real memory the block of memory addressed by the received logical address, and updating the plurality of logical partition valid bits in the entry corresponding to the block of memory addressed by the received logical address to indicate that the selected logical partition is contained in the real memory.

17. A method for address translation mapping of logical partitions for address translation buffer entries in a data processing system according to claim 13, wherein the step of combining is performed by concatenating the real block number with the plurality of logical partition selection bits and a real address offset to form a physical address for the selected partition of the memory reference.

18. A virtual memory mapping system for translating a logical address for a memory reference to a block of virtual memory into a physical address for the memory reference in real memory, wherein the block of virtual memory is divided into a plurality of logical partitions, and wherein a logical address includes a virtual block number addressing the block of memory in virtual memory space, a plurality of logical partition selection bits selecting the selected logical partition within the block of memory, and a real address offset, the virtual memory mapping system comprising:

an address translation buffer containing a plurality of entries, wherein each entry in the address translation buffer is associated with a block of virtual memory and includes a virtual block number, a real block number addressing the block of memory in real memory, and a plurality of logical partition valid bits, wherein a logical partition of the block of virtual memory is indicated as valid in real memory by an associated logical partition valid bit among the plurality of logical partition valid bits;

means for combining the real block number with the logical partition selection bits and the real address offset to form a physical address in real memory for the memory reference in the selected logical partition, when the selected logical partition is indicated as valid by the associated logical partition valid bit;

a comparator that receives a logical address for a memory reference to a selected logical partition of a block of virtual memory and determines if there is a match between the virtual block number of the received logical address and a virtual block number for any entry in the address translation buffer;

selection means for selecting the logical partition valid bit associated with the selected logical partition; and

a physical address register that receives and stores the real block number, the logical partition selection bits and the real address offset as a physical address to the real memory for the memory reference, when the logical partition valid bit associated with the selected logical partition indicates the selected partition is valid in real memory regardless of whether logical partitions of said particular block other than said selected logical partition are valid within said real memory.

19. A virtual memory mapping system according to claim 18, further comprising a main memory that outputs the memory reference in the selected logical partition addressed by the formed physical address.

20. A virtual memory mapping system according to claim 18, further comprising a main memory capable of storing the

selected partition of the block of memory addressed by the received logical address, and wherein the address translation buffer creates an entry for the block of memory addressed by the received logical address, when the comparator determines there is not a match of the virtual block number of the received logical address and the virtual block number of any entry in the address translation buffer.

21. A virtual memory mapping system according to claim **18**, further comprising a main memory capable of storing the block of memory addressed by the received logical address, and wherein the address translation buffer updates the plurality of logical partition valid bits in the entry corresponding to the block of memory addressed by the received logical address to indicate that the selected logical partition of the block of memory is contained in real memory, when the selected logical partition is not indicated as valid in an entry having a match with the virtual block number of the received logical address.

22. A virtual memory mapping system according to claim **18**, wherein the physical address register concatenates the real block number with the plurality of logical partition selection bits and a real address offset to form a physical address for the selected partition of the memory reference.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	:	5,708,790
DATED	:	Jan. 13, 1998
INVENTOR(S)	:	*Steven W. White et al.*

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

In col. 10, line 49, please **delete the number "4"** following "real".

In col. 11, line 37, please **delete "[of]"** following "bit".

Signed and Sealed this

Twenty-first Day of July, 1998

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US 20030009648A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2003/0009648 A1**

Doing et al. (43) **Pub. Date: Jan. 9, 2003**

(54) **APPARATUS FOR SUPPORTING A LOGICALLY PARTITIONED COMPUTER SYSTEM**

(75) Inventors: **Richard William Doing**, Rochester, MN (US); **Ronald Nick Kalla**, Zumbro Falls, MN (US); **Stephen Joseph Schwinn**, Lakeville, MN (US); **Edward John Silha**, Austin, TX (US); **Kenichi Tsuchiya**, Rochester, MN (US)

Correspondence Address:
IBM Corporation
Intellectual Property Law
Dept. 917
3605 Highway 52 North
Rochester, MN 55901 (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY

(21) Appl. No.: **10/175,626**

(22) Filed: **Jun. 20, 2002**

Related U.S. Application Data

(62) Division of application No. 09/346,206, filed on Jul. 1, 1999, now Pat. No. 6,438,671.

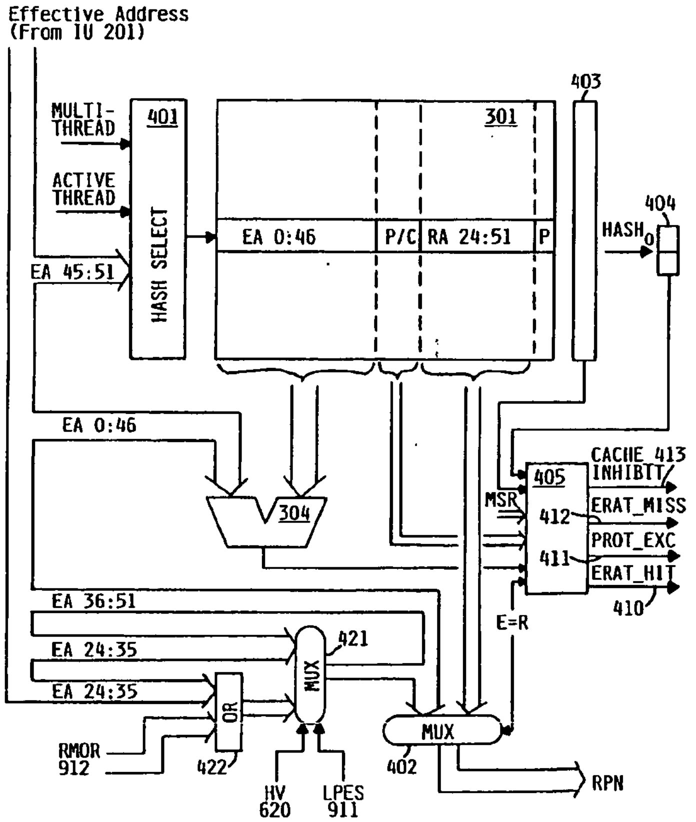
Publication Classification

(51) Int. Cl.^{7} **G06F 12/10**

(52) U.S. Cl. **711/202**; 711/173; 711/125; 709/100; 711/206

(57) **ABSTRACT**

A processor supports logical partitioning of hardware resources including real address spaces of a computer system. An ultra-privileged supervisor process, called a hypervisor, regulates the logical partitions and can dynamically re-allocate resources. Preferably, the processor supports hardware multithreading, each thread independently capable of being in either hypervisor, supervisor, or problem state. The processor assigns certain generated addresses to its logical partition, preferably by concatenating certain high order bits from a special register with lower order bits of the generated address. A separate range check mechanism concurrently verifies that these high order effective address bits are in fact 0, and generates an error signal if they are not.



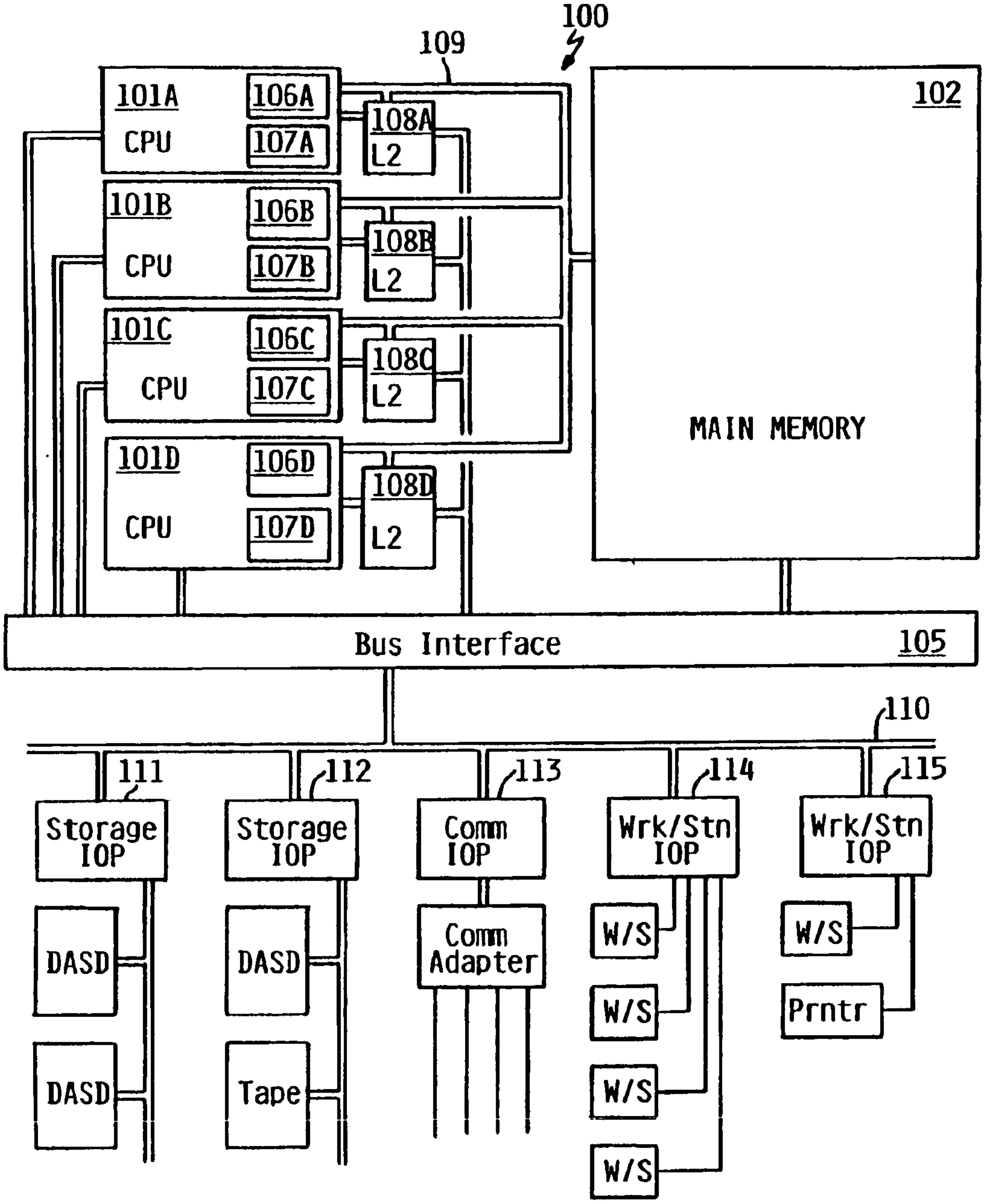

100
109
101A
CPU
106A
107A
108A
L2
101B
CPU
106B
107B
108B
L2
101C
CPU
106C
107C
108C
L2
101D
CPU
106D
107D
108D
L2
102
MAIN MEMORY
Bus Interface
105
110
111
112
113
114
115
Storage IOP
Storage IOP
Comm IOP
Wrk/Stn IOP
Wrk/Stn IOP
DASD
DASD
DASD
Tape
Comm Adapter
W/S
W/S
W/S
W/S
W/S
Prntr

FIG. 1

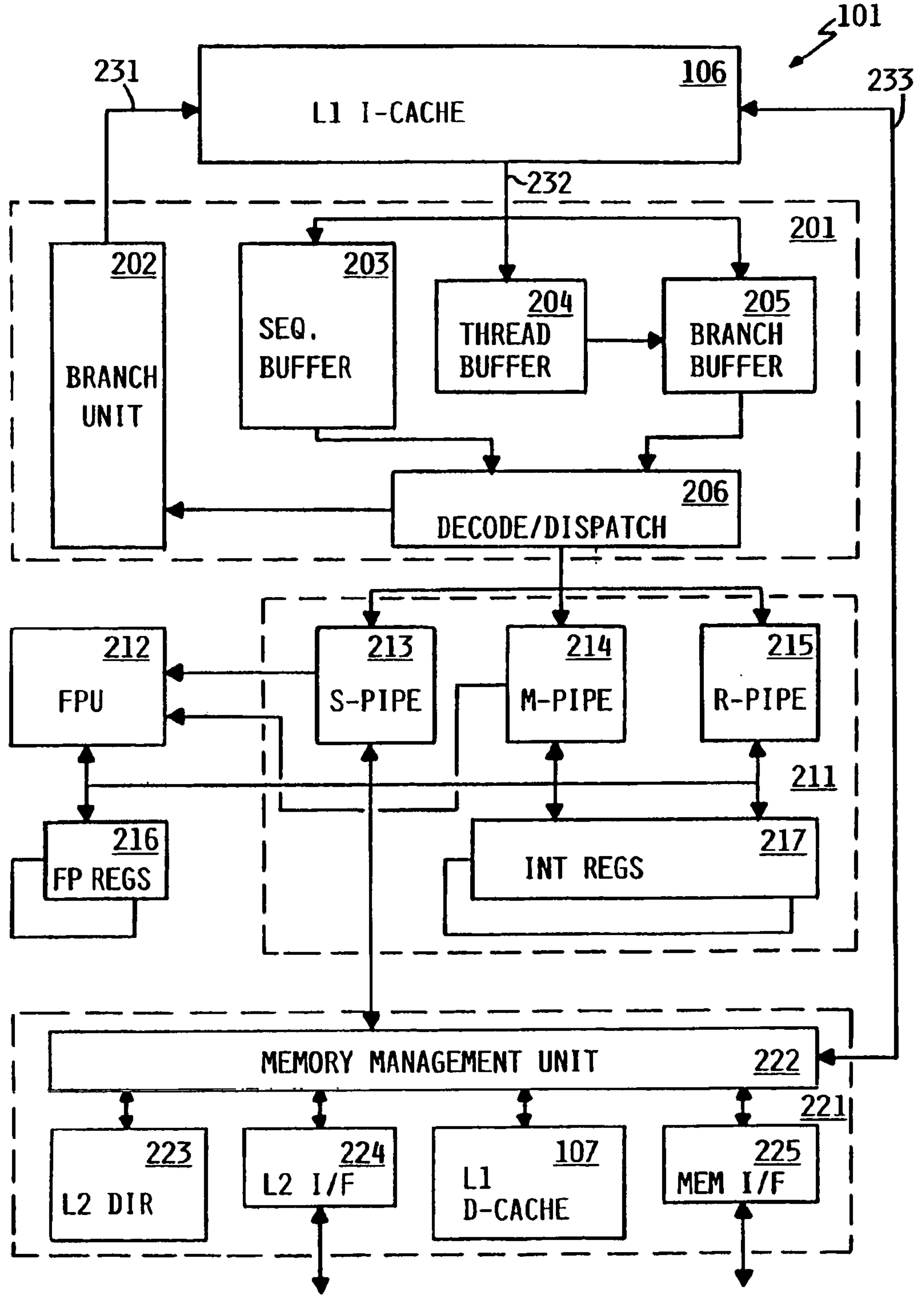

101
231
106
L1 I-CACHE
233
232
201
202
BRANCH UNIT
203
SEQ. BUFFER
204
THREAD BUFFER
205
BRANCH BUFFER
206
DECODE/DISPATCH
212
FPU
213
S-PIPE
214
M-PIPE
215
R-PIPE
211
216
FP REGS
217
INT REGS
MEMORY MANAGEMENT UNIT
222
221
223
L2 DIR
224
L2 I/F
107
L1 D-CACHE
225
MEM I/F

FIG. 2

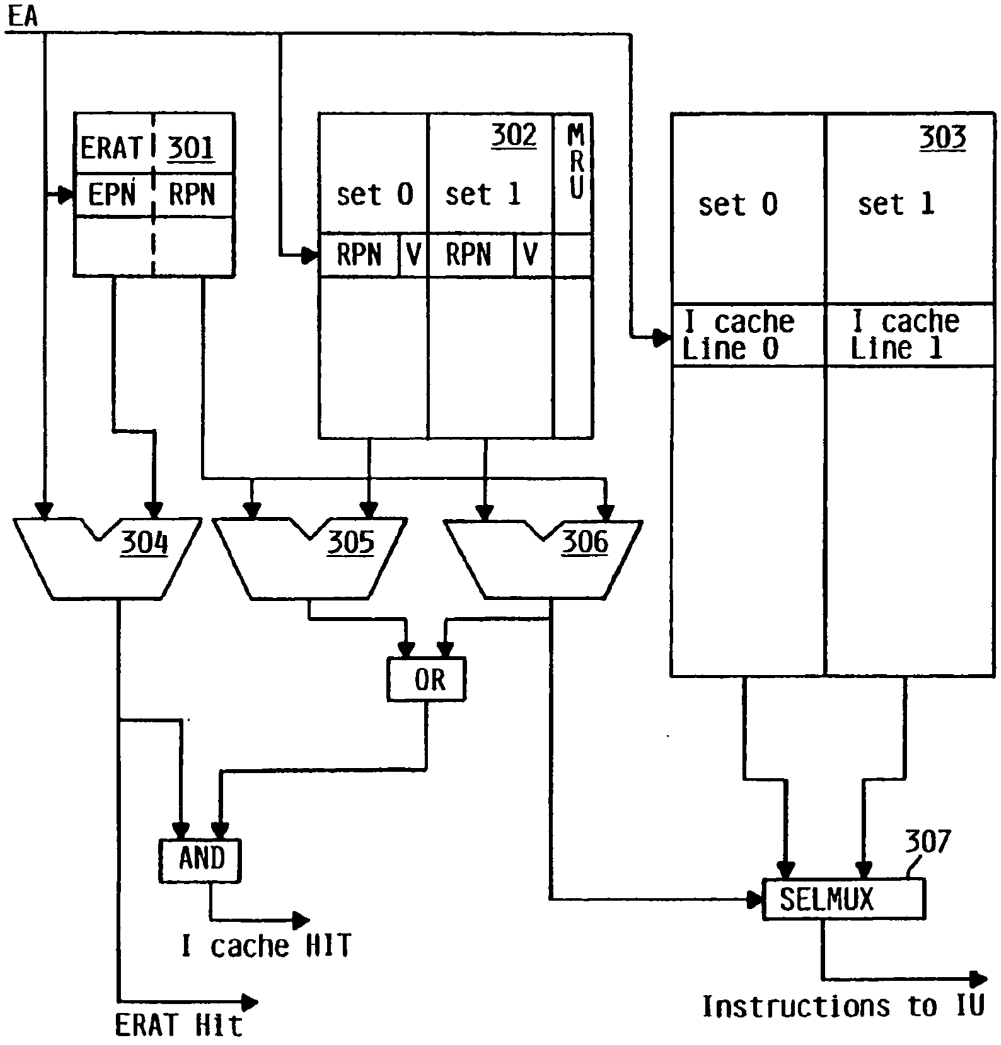

EA
ERAT
301
EPN
RPN
302
set 0
set 1
MRU
RPN
V
RPN
V
303
set 0
set 1
I cache Line 0
I cache Line 1
304
305
306
OR
AND
I cache HIT
ERAT Hit
307
SELMUX
Instructions to IU

FIG. 3

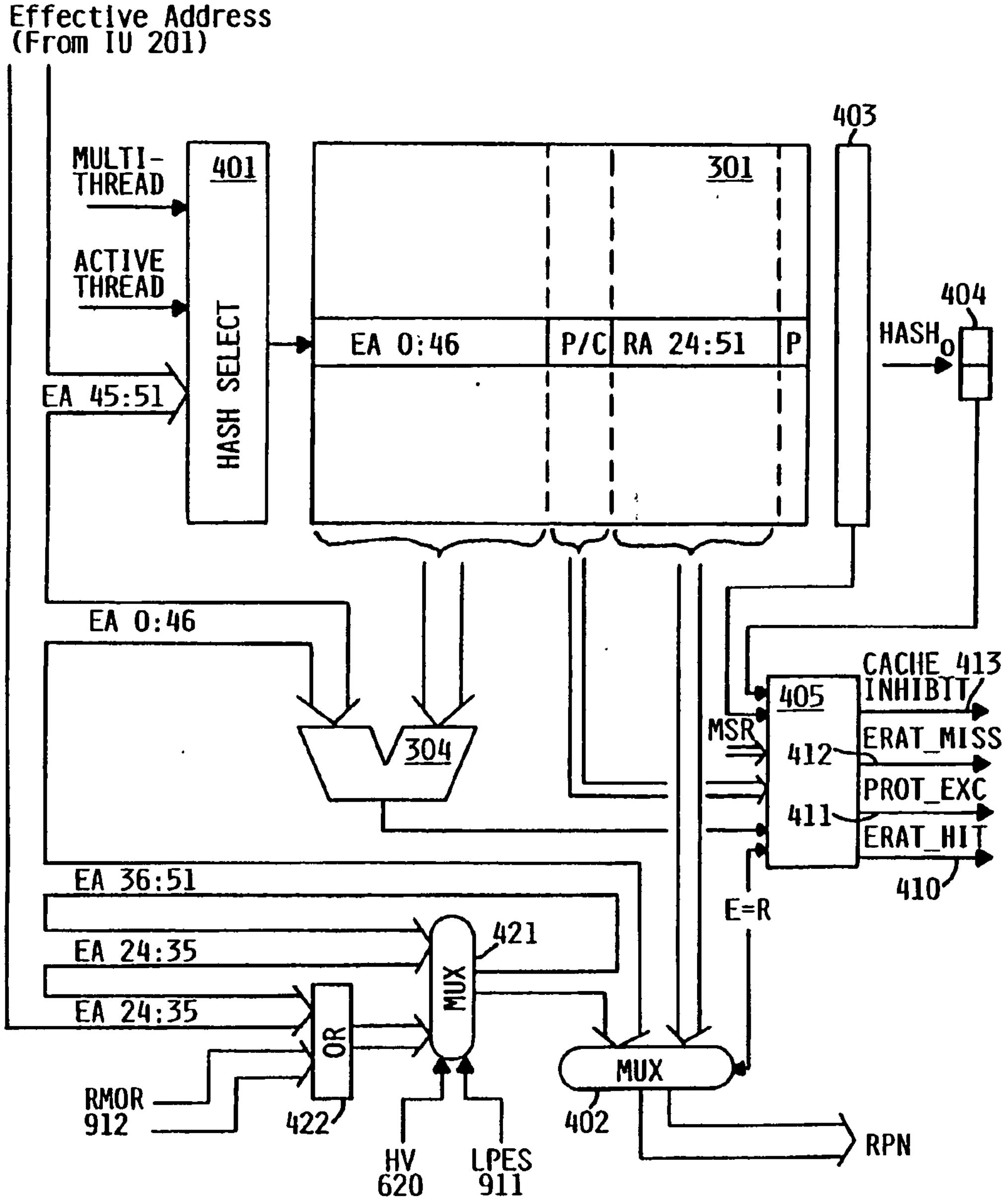

Effective Address
(From IU 201)
MULTI-
THREAD
ACTIVE
THREAD
EA 45:51
401
HASH SELECT
301
EA 0:46
P/C
RA 24:51
P
403
404
HASH$_0$
EA 0:46
304
MSR
405
412
411
CACHE 413
INHIBIT
ERAT_MISS
PROT_EXC
ERAT_HIT
410
E=R
EA 36:51
EA 24:35
EA 24:35
421
MUX
OR
RMOR
912
422
HV
620
LPES
911
MUX
402
RPN

FIG. 4

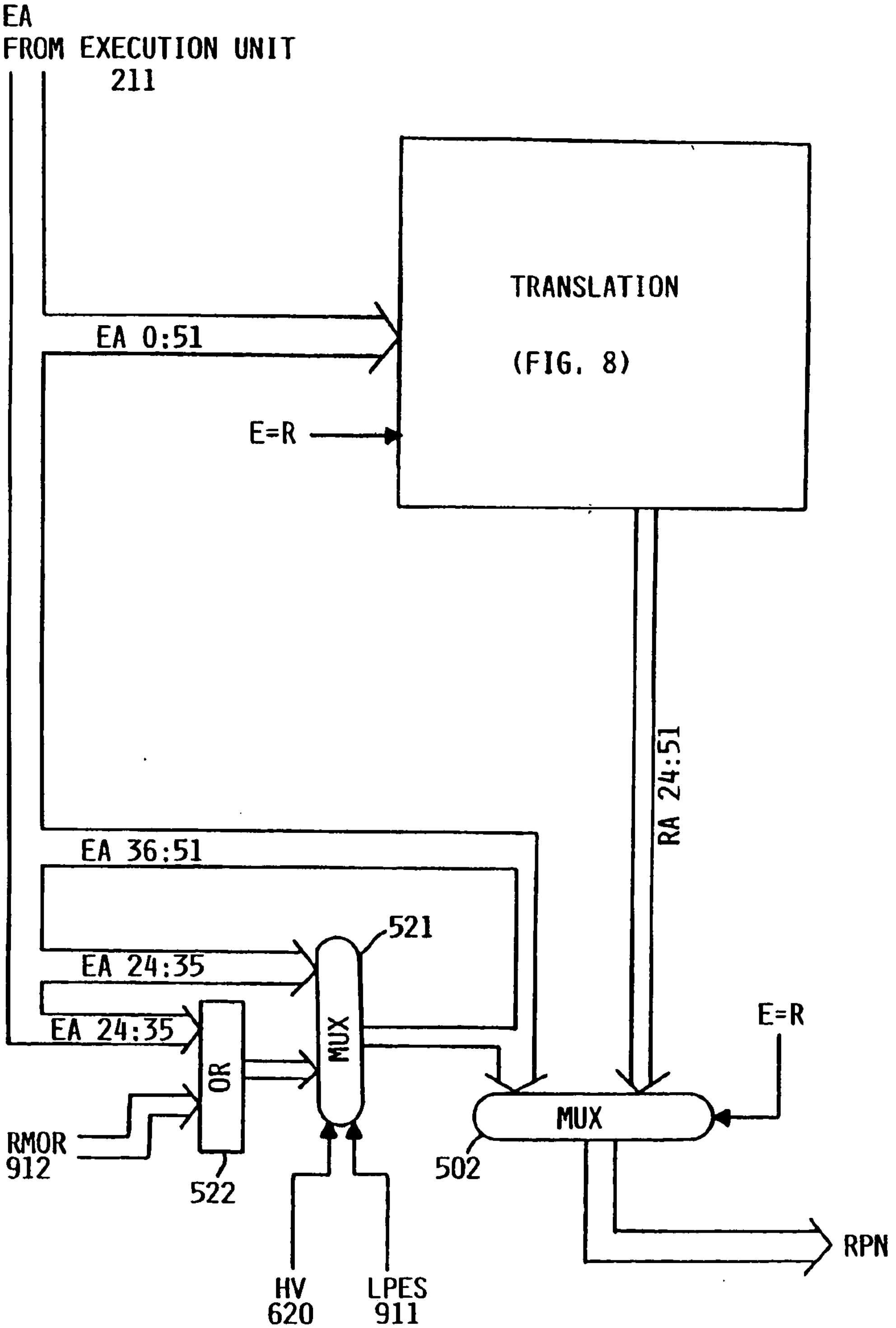

EA
FROM EXECUTION UNIT
211
TRANSLATION
(FIG. 8)
EA 0:51
E=R
RA 24:51
EA 36:51
521
EA 24:35
EA 24:35
MUX
OR
RMOR
912
522
HV
620
LPES
911
E=R
MUX
502
RPN

FIG. 5

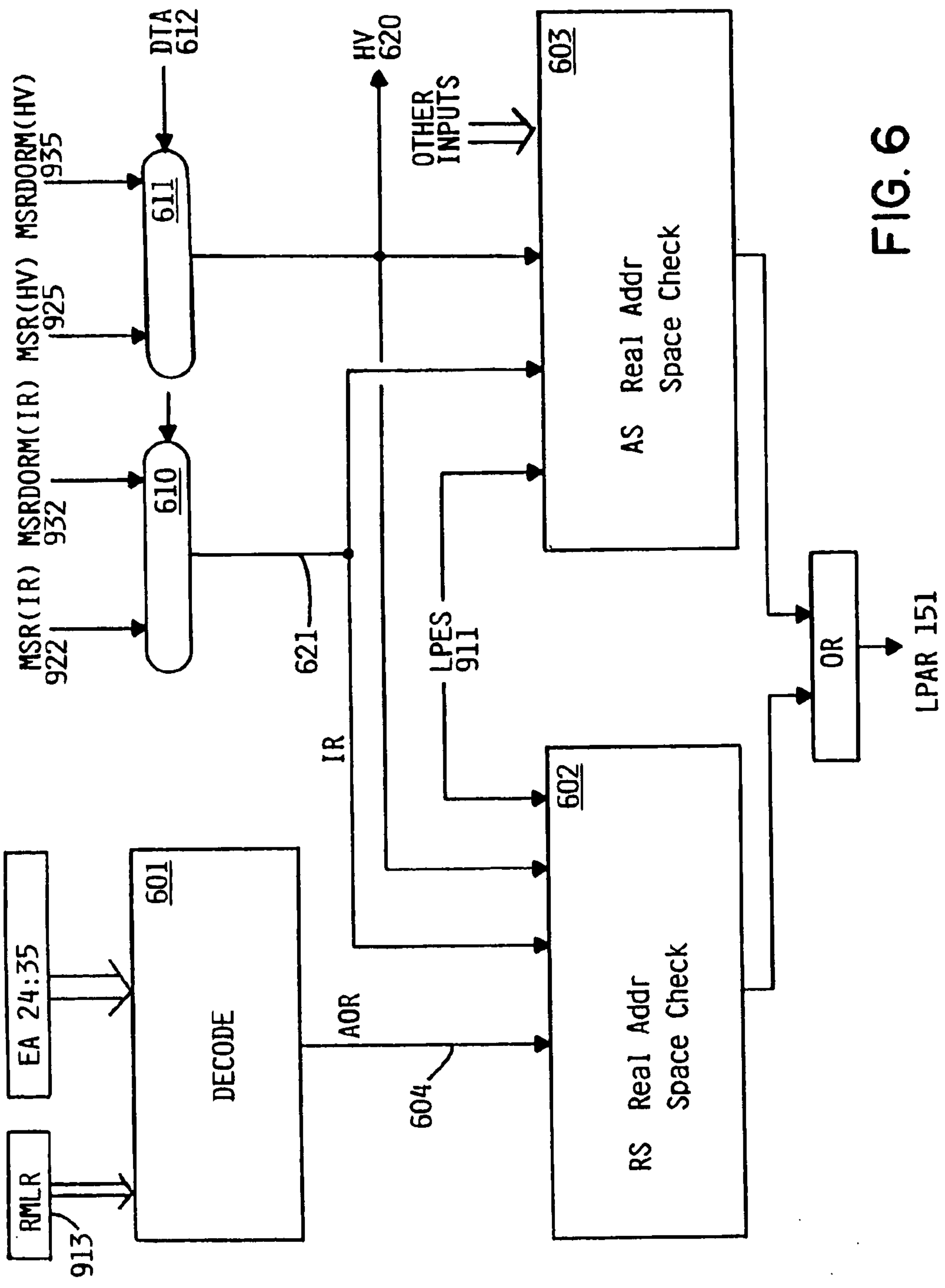

DTA
612
HV
620
MSRDORM(HV)
935
MSR(HV)
925
MSRDORM(IR)
932
MSR(IR)
922
611
610
621
IR
OTHER INPUTS
603
AS Real Addr Space Check
LPES
911
602
RS Real Addr Space Check
OR
LPAR 151
EA 24:35
RMLR
913
601
DECODE
AOR
604

FIG. 6

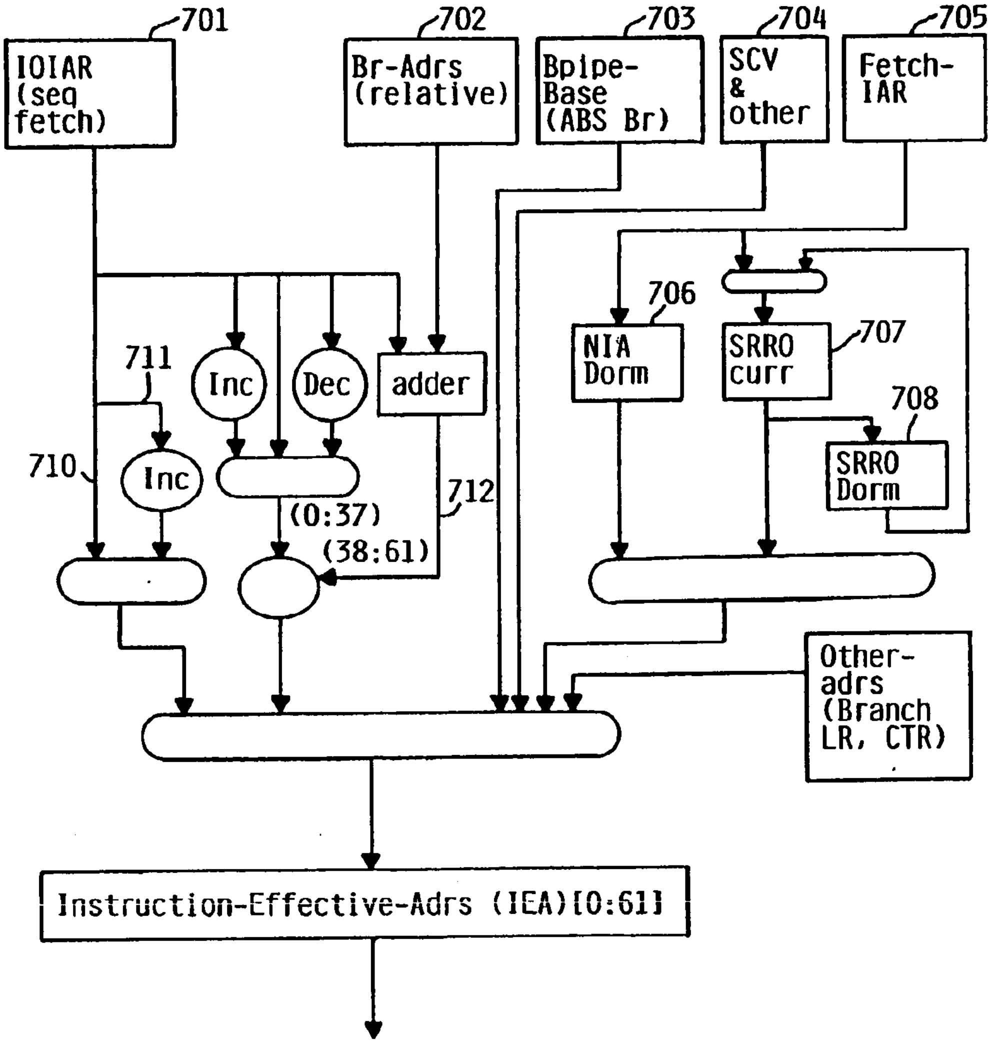

701
IOIAR (seq fetch)
702
Br-Adrs (relative)
703
Bpipe-Base (ABS Br)
704
SCV & other
705
Fetch-IAR
706
NIA Dorm
707
SRRO curr
708
SRRO Dorm
711
710
Inc
Inc
Dec
adder
712
(0:37)
(38:61)
Other-adrs (Branch LR, CTR)
Instruction-Effective-Adrs (IEA)[0:61]

FIG. 7

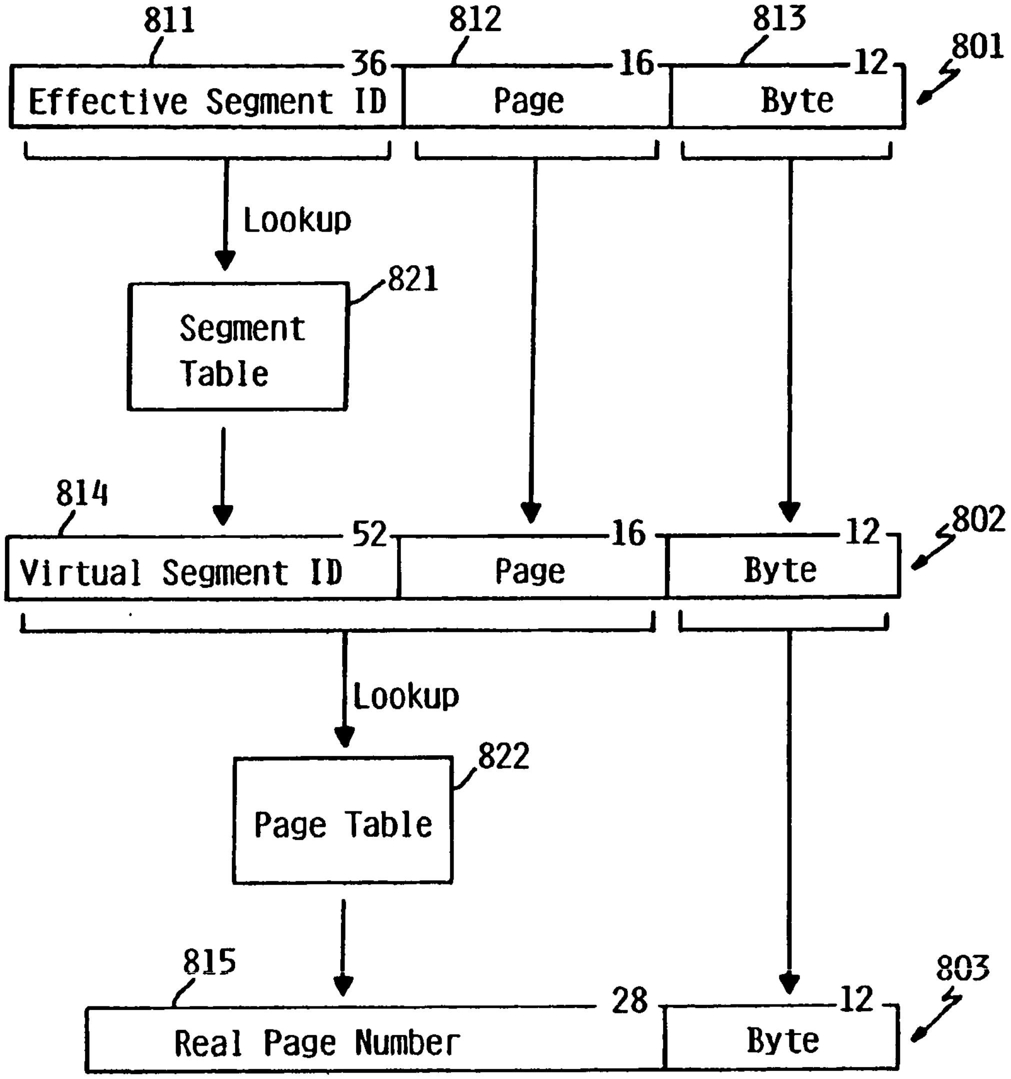

811
36
Effective Segment ID
812
16
Page
813
12
Byte
801
Lookup
Segment Table
821
814
52
Virtual Segment ID
16
Page
12
Byte
802
Lookup
Page Table
822
815
28
Real Page Number
12
Byte
803

FIG. 8

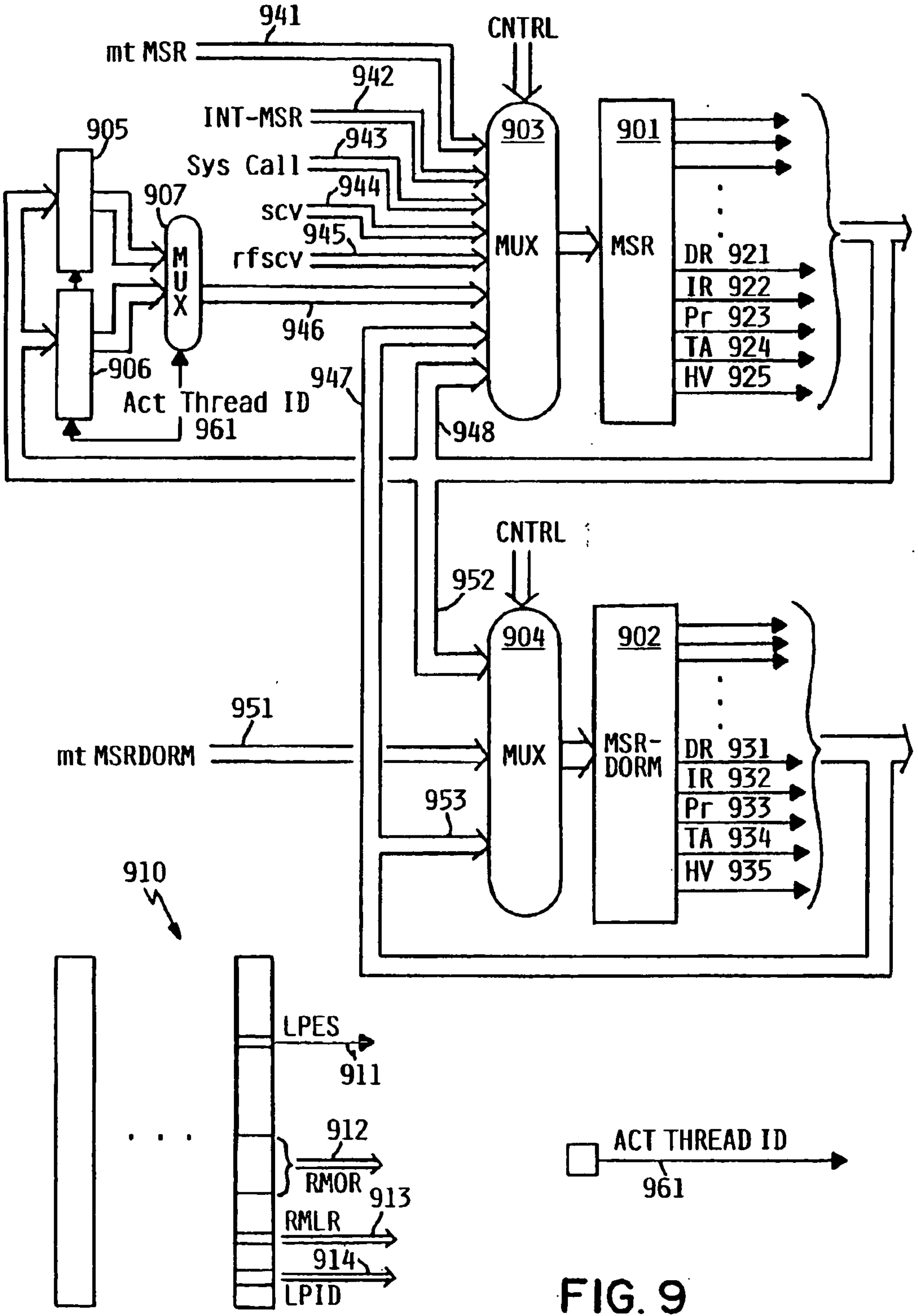

941
mt MSR
CNTRL
942
INT-MSR
903
901
905
943
Sys Call
944
907
scv
945
rfscv
MUX
MSR
M
U
X
DR 921
IR 922
Pr 923
TA 924
HV 925
946
906
947
Act Thread ID
961
948
CNTRL
952
904
902
951
mt MSRDORM
MUX
MSR-
DORM
DR 931
IR 932
Pr 933
TA 934
HV 935
953
910
LPES
911
912
RMOR
913
RMLR
914
LPID
ACT THREAD ID
961

FIG. 9

APPARATUS FOR SUPPORTING A LOGICALLY PARTITIONED COMPUTER SYSTEM

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This is a divisional application of U.S. patent application Ser. No. 09/346,206, filed Jul. 1, 1999, originally entitled "APPARATUS FOR SUPPORTING A LOGICALLY PARTITIONED COMPUTER SYSTEM", and by subsequent amendment entitled "GENERATING PARTITION CORRESPONDING REAL ADDRESS IN PARTITIONED MODE SUPPORTING SYSTEM", which is herein incorporated by reference.

[0002] The present application is also related to the following commonly assigned U.S. patents and patent applications, all of which are herein incorporated by reference:

[0003] Ser. No. 09/314,769, filed May 19, 1999, entitled Processor Reset Generated Via Memory Access Interrupt (Assignee's docket no. R0999-022).

[0004] Ser. No. 09/314,541, filed May 19, 1999, entitled Apparatus and Method for Specifying Maximum Interactive Performance in a Logical Partition of a Computer (Assignee's docket no. R0999-021).

[0005] Ser. No. 09/314,324, filed May 19, 1999, entitled Management of a Concurrent Use License in a Logically Partitioned Computer (Assignee's docket no. R0999-023).

[0006] Ser. No. 09/314,214, filed May 19, 1999, entitled Logical Partition Manager and Method (Assignee's docket no. R0999-025).

[0007] U.S. Pat. No. 6,279,046 to Armstrong et al., entitled Event-Driven Communications Interface for Logically-Partitioned Computer.

[0008] U.S. Pat. No. 6,161,166 to Doing et al., entitled Instruction Cache for Multithreaded Processor.

[0009] U.S. Pat. No. 6,263,404 to Borkenhagen et al., entitled Accessing Datafrom a Multiple Entry Fully Associative Cache Buffer in a Multithread Data Processing System.

[0010] U.S. Pat. No. 6,021,481 to Eickemeyer et al., entitled Effective-To-RealAddress Cache Managing Apparatus and Method.

[0011] U.S. Pat. No. 6,212,544 to Borkenhagen et al., entitled Altering Thread Priorities in a Multithreaded Processor.

[0012] Ser. No. 08/958,716, filed Oct. 23, 1997, entitled Method and Apparatus for Selecting Thread Switch Events in a Multithreaded Processor (Assignee's docket no. R0997-104).

[0013] Ser. No. 08/957,002, filed Oct. 23, 1997, entitled Thread Switch Control in a Multithreaded Processor System (Assignee's docket no. R0996-042).

[0014] U.S. Pat. No. 6,105,051 to Borkenhagen et al., entitled An Apparatus and Method to Guarantee Forward Progress in a Multithreaded Processor.

[0015] U.S. Pat. No. 6,076,157 to Borkenhagen et al., entitled Method and Apparatus To Force a Thread Switch in a Multithreaded Processor.

[0016] U.S. Pat. No. 6,088,788 to Borkenhagen et al., entitled Background Completion of Instruction and Associated Fetch Request in a Multithread Processor.

FIELD OF THE INVENTION

[0017] The present invention relates generally to digital data processing, and more particularly to support within a processing unit for logically partitioning of a digital computer system.

BACKGROUND OF THE INVENTION

[0018] A modem computer system typically comprises a central processing unit (CPU) and supporting hardware necessary to store, retrieve and transfer information, such as communications busses and memory. It also includes hardware necessary to communicate with the outside world, such as input/output controllers or storage controllers, and devices attached thereto such as keyboards, monitors, tape drives, disk drives, communication lines coupled to a network, etc. The CPU is the heart of the system. It executes the instructions which comprise a computer program and directs the operation of the other system components.

[0019] From the standpoint of the computer's hardware, most systems operate in fundamentally the same manner. Processors are capable of performing a limited set of very simple operations, such as arithmetic, logical comparisons, and movement of data from one location to another. But each operation is performed very quickly. Programs which direct a computer to perform massive numbers of these simple operations give the illusion that the computer is doing something sophisticated. What is perceived by the user as a new or improved capability of a computer system is made possible by performing essentially the same set of very simple operations, but doing it much faster. Therefore continuing improvements to computer systems require that these systems be made ever faster.

[0020] The overall speed of a computer system (also called the "throughput") may be crudely measured as the number of operations performed per unit of time. Conceptually, the simplest of all possible improvements to system speed is to increase the clock speeds of the various components, and particularly the clock speed of the processor. E.g., if everything runs twice as fast but otherwise works in exactly the same manner, the system will perform a given task in half the time. Early computer processors, which were constructed from many discrete components, were susceptible to significant speed improvements by shrinking component size, reducing component number, and eventually, packaging the entire processor as an integrated circuit on a single chip. The reduced size made it possible to increase the clock speed of the processor, and accordingly increase system speed.

[0021] Despite the enormous improvement in speed obtained from integrated circuitry, the demand for ever faster computer systems has continued. Hardware designers have been able to obtain still further improvements in speed by greater integration (i.e., increasing the number of circuits packed onto a single chip), by further reducing the size of the circuits, and by various other techniques. However, designers can see that physical size reductions can not continue indefinitely, and there are limits to their ability to continue to increase clock speeds of processors. Attention has therefore been directed to other approaches for further improvements in overall speed of the computer system.

[0022] Without changing the clock speed, it is possible to improve system throughput by using multiple processors. The modest cost of individual processors packaged on integrated circuit chips has made this practical. While there are certainly potential benefits to using multiple processors, numerous additional architectural issues are introduced. In particular, multiple processors typically share the same main memory (although each processor may have it own cache). It is necessary to devise mechanisms that avoid memory access conflicts. For example, if two processors have the capability to concurrently read and update the same data, there must be mechanisms to assure that each processor has authority to access the data, and that the resulting data is not gibberish. Without delving into further architectural complications of multiple processor systems, it can still be observed that there are many reasons to improve the speed of the individual CPU, whether or not a system uses multiple CPUs or a single CPU. If the CPU clock speed is given, it is possible to further increase the speed of the individual CPU, i.e., the number of operations executed per second, by increasing the average number of operations executed per clock cycle.

[0023] In order to boost CPU speed, it is common in high performance processor designs to employ instruction pipelining, as well as one or more levels of cache memory. Pipeline instruction execution allows subsequent instructions to begin execution before previously issued instructions have finished. Cache memories store frequently used and other data nearer the processor and allow instruction execution to continue, in most cases, without waiting the full access time of a main memory.

[0024] Pipelines will stall under certain circumstances. An instruction that is dependent upon the results of a previously dispatched instruction that has not yet completed may cause the pipeline to stall. For instance, instructions dependent on a load/store instruction in which the necessary data is not in the cache, i.e., a cache miss, cannot be executed until the data becomes available in the cache. Maintaining the requisite data in the cache necessary for continued execution and to sustain a high hit ratio, i.e., the number of requests for data compared to the number of times the data was readily available in the cache, is not trivial especially for computations involving large data structures. A cache miss can cause the pipelines to stall for several cycles, and the total amount of memory latency will be severe if the data is not available most of the time. Although memory devices used for main memory are becoming faster, the speed gap between such memory chips and high-end processors is becoming increasingly larger. Accordingly, a significant amount of execution time in current high-end processor designs is spent waiting for resolution of cache misses.

[0025] It can be seen that the reduction of time the processor spends waiting for some event, such as re-filling a pipeline or retrieving data from memory, will increase the average number of operations per clock cycle. One architectural innovation directed to this problem is called "multithreading". This technique involves breaking the workload into multiple independently executable sequences of instructions, called threads. At any instant in time, the CPU maintains the state of multiple threads. As a result, it is relatively simple and fast to switch threads. The term "multithreading" as defined in the computer architecture community is not the same as the software use of the term which means one task subdivided into multiple related threads. In the architecture definition, the threads may be independent. Therefore "hardware multithreading" is often used to distinguish the two uses of the term. As used herein, "multithreading" will refer to hardware multithreading.

[0026] There are two basic forms of multithreading. In the more traditional form, sometimes called "fine-grained multithreading", the processor executes N threads concurrently by interleaving execution on a cycle-by-cycle basis. This creates a gap between the execution of each instruction within a single thread, which removes the need for the processor to wait for certain short term latency events, such as re-filling an instruction pipeline. In the second form of multithreading, sometimes called "coarse-grained multithreading", multiple instructions in a single thread are sequentially executed until the processor encounters some longer term latency event, such as a cache miss.

[0027] Typically, multithreading involves replicating the processor registers for each thread in order to maintain the state of multiple threads. For instance, for a processor implementing the architecture sold under the trade name PowerPC™ to perform multithreading, the processor must maintain N states to run N threads. Accordingly, the following are replicated N times: general purpose registers, floating point registers, condition registers, floating point status and control register, count register, link register, exception register, save/restore registers, and special purpose registers. Additionally, the special buffers, such as a segment lookaside buffer, can be replicated or each entry can be tagged with the thread number and, if not, must be flushed on every thread switch. Also, some branch prediction mechanisms, e.g., the correlation register and the return stack, should also be replicated. However, larger hardware structures such as caches and execution units are typically not replicated.

[0028] In a computer system using multiple CPUs (symmetrical multi-processors, or SMPs), each processor supporting concurrent execution of multiple threads, the enforcement of memory access rules is a complex task. In many systems, each user program is granted a discrete portion of address space, to avoid conflicts with other programs and prevent unauthorized accesses. However, something must allocate addresses in the first place, and perform other necessary policing functions. Therefore, special supervisor programs exist which necessarily have access to the entire address space. It is assumed that these supervisor programs contain "trusted" code, which will not disrupt the operation of the system. In the case of a multiprocessor system, it is possible that multiple supervisor programs will be running on multiple SMPs, each having extraordinary capability to access data addresses in memory. While this does not necessarily mean that data will be corrupted or compromised, avoidance of potential problems adds another layer of complexity to the supervisor code. This additional complexity can adversely affect system performance. To the extent hardware within each SMP can assist software supervisors, performance can be improved.

[0029] In a large multiprocessor system, it may be desirable to partition the system into one or more smaller logical SMPs, an approach known as logical partitioning. In addition, once a system is partitioned it may be desirable to dynamically re-partition the system based on changing requirements. It is possible to do this using only software. The additional complexity this adds to the software can adversely affect system performance. Logical partitioning of a system would be more effective if hardware support were provided to assist the software. Hardware support may be useful to help software isolate one logical partition from another. Said differently, hardware support may be used to prevent work being performed in one logical partition from corrupting work being performed in another. Hardware support would also be useful for dynamically re-partitioning the system in an efficient manner. This hardware support may be used to enforce the partitioning of system resources such as processors, real memory, internal registers, etc.

SUMMARY OF THE INVENTION

[0030] It is therefore an object of the present invention to provide an improved processor apparatus.

[0031] Another object of this invention is to provide greater support, and in particular hardware support, for logical partitioning of a computer system.

[0032] Another object of this invention is to provide an apparatus having greater hardware regulation of memory access in a processor.

[0033] Another object of this invention is to increase the performance of a computer system having multiple processors.

[0034] Another object of the invention is to improve multithreaded processor hardware control for logical partitioning of a computer system.

[0035] A processor provides hardware support for logical partitioning of a computer system. Logical partitions isolate the real address spaces of processes executing on different processors, specifically, supervisory processes. An ultra-privileged supervisor process, called a hypervisor, regulates the logical partitions.

[0036] In the preferred embodiment, the processor contains multiple register sets for supporting the concurrent execution of multiple threads (i.e., hardware multithreading). Each thread is capable of independently being in either hypervisor, supervisor or problem (non-privileged) state.

[0037] In the preferred embodiment, each processor generates effective addresses from executable code, which are translated to real addresses corresponding to locations in physical main memory. Certain processes, particularly supervisory processes, may optionally run in a special (effective address equals real address) mode. In this mode, real addresses are constrained within a logical partition by effectively concatenating certain high order bits from a special register (real memory offset register) with lower order bits of the effective address. For clarity, the effective address in effective=real mode is referred to herein as a base real address, while the resultant address after partitioning is referred to as a partitioned real address. Logical partitioning of the address space amounts to an enforced constraint on certain high order address bits, so that within any given partition these address bits are the same. Partitioning is thus distinguished from typical address translation, wherein a range of effective addresses is arbitrarily correlated a range of real addresses. The hardware which partitions a real address is actually a set of OR gates which perform a logical OR of the contents of the real memory offset register with an equal number of high order bits of effective address (base real address). By convention, the high order bits of effective address (i.e., in the base real address) which are used constrain the address to a logical partition should be 0. A separate range check mechanism concurrently verifies that these high order effective address bits are in fact 0, and generates a real address space check signal if they are not.

[0038] In the preferred embodiment, the range check mechanism includes a 2-bit real memory limit register, and a set of logic gates. The limit register specifies the number of high order effective address (base real address) bits which must be zero (i.e., the size of the logical partition memory resource). The limit register value generates a mask, which is logically ANDed with selected bits of the effective address. The resulting bits are then logically ORed together to generate the real address space check signal. The use of this limit register mechanism supports logically partitioned memory spaces of different sizes.

[0039] In the preferred embodiment, instruction addresses can be pre-fetched in anticipation of execution. In particular, dormant thread instructions may be pre-fetched while another thread is processing and executing instructions. The partitioning mechanism checks and controls instruction pre-fetching independently of the actively running thread.

dynamically reallocate

[0040] In the preferred embodiment, special operating system software running in hypervisor state can dynamically re-allocate resources to logical partitions. In particular, it can alter the contents of the real memory offset register and the real memory limit register which regulate the generation of partitioned real addresses; a logical partition identifier which identifies the logical partition to which a processor is assigned; and certain configuration information.

1A

1B

[0041] In the preferred embodiment, the processor supports different systems which use the hypervisor, supervisor and problem states differently. Thus, one mode of operation supports effective=real addressing mode in any state, but addresses are partitioned and checked as described above when operating in non-hypervisor state. A second mode of operation supports effective=real addressing mode in only the hypervisor state.

[0042] The enforcement of logical partitioning by processor hardware which intercepts a base real address and converts it to a partitioned real address removes the need for low-level operating system software to verify certain address constraints among multiple processors and threads, reducing the burden on operating system software and improving system performance.

[0043] Other objects, features and characteristics of the present invention; methods, operation, and functions of the related elements of the structure; combination of parts; and the like will become apparent from the following detailed description of the preferred embodiments and accompanying drawings, all of which form a part of this specification, wherein like reference numerals designate corresponding parts in the various figures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0044] **FIG. 1** is a high-level block diagram of the major hardware components of a computer system having multiple CPUs, according to the preferred embodiment of the invention described herein.

[0045] **FIG. 2** is a high-level diagram of a central processing unit of a computer system according to the preferred embodiment.

[0046] **FIG. 3** illustrates the major components of an L1 instruction cache, according to the preferred embodiment.

[0047] **FIG. 4** illustrates in greater detail real address partitioning logic, an effective to real address table and associated control structures for instruction addresses, according to the preferred embodiment.

[0048] **FIG. 5** illustrates real address partitioning logic for data addresses, according to the preferred embodiment.

[0049] **FIG. 6** illustrates the generation of instruction storage interrupts for enforcing logical partitioning, according to the preferred embodiment.

[0050] **FIG. 7** illustrates at a high level the generation of effective addresses for instructions, according to the preferred embodiment.

[0051] **FIG. 8** is a logical illustration of address translation, according to the preferred embodiment.

[0052] **FIG. 9** illustrates the operation of certain state and configuration registers, according to the preferred embodiment.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0053] The major hardware components of a multiprocessor computer system **100** for utilizing the logical partitioning architecture according to the preferred embodiment of the present invention are shown in **FIG. 1**. CPUs **101A**, **101B**, **101C** and **101D** for processing instructions contains separate respective internal level one instruction caches **106A**, **106B**, **106C**, **106D** (L1I-cache) and level one data caches **107A**, **107B**, **107C**, **107D** (L1D-cache). Each L1I-cache **106A**, **106B**, **106C**, **106D** stores instructions for execution by its CPU **101A**, **101B**, **101C**, **101D**. L I D-cache stores data (other than instructions) to be processed by a CPU. Each CPU **101A**, **101B**, **101C**, **101D** is coupled to a respective level two cache (L2 cache) **108A**, **108B**, **108C**, **108D**, which can be used to hold both instructions and data. Memory bus **109** transfers data between L2 caches or CPU on the one hand and main memory **102** on the other. CPUs **101A**, **101B**, **101C**, **101D**, L2 cache **108A**, **108B**, **108C**, **108D** and main memory **102** also communicate via bus interface **105** with system bus **110**. Various I/O processing units (IOPs) **111-115** attach to system bus **110** and support communication with a variety of storage and I/O devices, such as direct access storage devices (DASD), tape drives, workstations, printers, and remote communication lines for communicating with remote devices or other computer systems. For simplicity, CPU, L1I-cache, L1D-cache, and L2 cache are herein designated generically by reference numbers **101**, **106**, **107** and **108**, respectively. While various buses are shown in **FIG. 1**, it should be understood that these are intended to represent various communications paths at a conceptual level, and that the actual physical configuration of buses may vary.

[0054] In the preferred embodiment, each CPU is capable of maintaining the state of two threads, and switches execution between threads on certain latency events. I.e., CPU executes a single thread (the active thread) until some latency event is encountered which would force the CPU to wait, (a form of coarse-grained multithreading). Thread switching conditions and mechanisms are described in greater detail in U.S. Pat. No. 6,212,544, U.S. Pat. No. 6,105,051, U.S. Pat. No. 6,076,157, and in commonly assigned copending U.S. patent application Ser. Nos. 08/958,716, 08/957,002, all filed Oct. 23, 1997, and incorporated herein by reference. However, it should be understood that the present invention could be practiced with a different number of thread states in each CPU, and that it would be possible to interleave execution of instructions from each thread on a cycle-by-cycle basis (fine-grained multithreading), or to switch threads on some different basis.

[0055] **FIG. 2** is a high level diagram of the major components of CPU **101**, showing CPU **101** in greater detail than is depicted in **FIG. 1**, according to the preferred embodiment. In this embodiment, the components shown in **FIG. 2** are packaged on a single semiconductor chip. CPU **101** includes instruction unit portion **201**, execution unit portion **211** and **212**, and storage control portion **221**. In general, instruction unit **201** obtains instructions from L1I-cache **106**, decodes instructions to determine operations to perform, and resolves branch conditions to control program flow. Execution unit **211** performs arithmetic and logical operations on data in registers, and loads or stores data. Storage control unit **221** accesses data in the L1 data cache or interfaces with memory external to the CPU where instructions or data must be fetched or stored.

[0056] Instruction unit **201** comprises branch unit **202**, buffers **203**, **204**, **205**, and decode/dispatch unit **206**. Instructions from L1I-cache **106** are loaded into one of the three buffers from L1I-Cache instruction bus **232**. Sequential buffer **203** stores 16 instructions in the current execution sequence. Branch buffer **205** stores 8 instructions from a branch destination; these are speculatively loaded into buffer **205** before branch evaluation, in the event the branch is taken. Thread switch buffer **204** stores 8 instructions for the inactive thread; in the event a thread switch is required from the currently active to the inactive thread, these instructions will be immediately available. Decode/dispatch unit **206** receives the current instruction to be executed from one of the buffers, and decodes the instruction to determine the operation(s) to be performed or branch conditions. Branch unit **202** controls the program flow by evaluating branch conditions, and refills buffers from L1I-cache **106** by sending an effective address of a desired instruction on L1I-Cache address bus **231**.

[0057] Execution unit **211** comprises S-pipe **213**, M-pipe **214**, R-pipe **215** and a bank of general purpose registers **217**. Registers **217** are divided into two sets, one for each thread. R-pipe is a pipelined arithmetic unit for performing a subset of integer arithmetic and logic functions for simple integers. M-pipe **214** is a pipelined arithmetic unit for performing a larger set of arithmetic and logic functions. S-pipe **213** is a pipelined unit for performing load and store operations.

Floating point unit 212 and associated floating point registers 216 are used for certain complex floating point operations which typically require multiple cycles. Like general purpose registers 217, floating point registers 216 are divided into two sets, one for each thread.

[0058] Storage control unit 221 comprises memory management unit 222, L2 cache directory 223, L2 cache interface 224, L1 data cache 107, and memory bus interface 225. L1 D-cache is an on-chip cache used for data (as opposed to instructions). L2 cache directory 223 is a directory of the contents of L2 cache 108. L2 cache interface 224 handles the transfer of data directly to and from L2 cache 108. Memory bus interface 225 handles the transfer of data across memory bus 109, which may be to main memory 102 or to L2 cache units associated with other CPUs. Memory management unit 222 is responsible for routing data accesses to the various units. E.g., when S-pipe 213 processes a load command, requiring data to be loaded to a register, memory management unit may fetch the data from L1D-cache 107, L2 cache 108, or main memory 102. Memory management unit 222 determines where to obtain the data. L1D-cache 107 is directly accessible, as is the L2 cache directory 223, enabling unit 222 to determine whether the data is in either L1D-cache 107 or L2 cache 108. If the data is in neither on-chip L1D-cache nor L2 cache 108, it is fetched from memory bus 109 using memory interface 225.

[0059] While various CPU components have been described and shown at a high level, it should be understood that the CPU of the preferred embodiment contains many other components not shown, which are not essential to an understanding of the present invention. For example, various additional special purpose registers will be required in a typical design, some of which must be replicated for each thread. It should also be understood that the number, type and arrangement of components within CPU 101 could be varied. For example, the number and configuration of buffers and caches may vary; the number and function of execution unit pipelines may vary; registers may be configured in different arrays and sets; dedicated floating point processing hardware may or may not be present; etc.

[0060] CPU 101 of the preferred embodiment supports multiple levels of address translation, as logically illustrated in FIG. 8. The three basic addressing constructs are effective address 801, virtual address 802, and real address 803. An "effective address" refers to the address from the point of view of the executable code, i.e., it is an instruction address generated by instruction unit 201, or a data address generated by execution unit 211. An effective address may be produced in any of various ways known in the art, e.g., as a concatenation of some high-order address bits in a special-purpose register (which changes infrequently, e.g., when execution of a new task is initiated) and lower order address bits from an instruction; as a computed offset from an address in a general purpose register; as an offset from the currently executing instruction; etc., as illustrated in greater detail in FIG. 7, explained below. In this embodiment, an effective address comprises 64 bits, numbered 0 to 63 (0 being the highest order bit). A "virtual address" is an operating system construct, used to isolate the address spaces of different users. I.e., if each user may reference the full range of effective addresses, then the effective address spaces of different users must be mapped into a larger virtual address space to avoid conflicts. The virtual address is not a physical entity in the sense that it is stored in registers; it is a logical construction, resulting from a concatenation of a 52-bit virtual segment ID 814 and the low-order 28 bits of the effective address, a total of 80 bits. A "real address" refers to a physical location in memory 102 where the instruction or data is stored. The real address comprises 40 bits numbered 24 to 63 (24 being the highest order bit).

[0061] As shown in FIG. 8, an effective address 801 comprises 36-bit effective segment ID 811, 16-bit page number 812, and 12-bit byte index 813, the effective segment ID occupying the highest order bit positions. A virtual address 802 is constructed from an effective address by mapping the 36-bit effective segment ID 811 to a 52-bit virtual segment ID 814, and concatenating the resultant virtual segment ID 814 with page number 812 and byte index 813. A real address 803 is derived from the virtual address by mapping the virtual segment ID 814 and page number 812 to a 28-bit real page number 815, and concatenating the real page number with byte index 813. Because a page of main memory contains 4K (i.e., 2^{12}) bytes, the byte index 813 (lowest order 12 address bits) specifies an address within a page, and is the same whether the address is effective, virtual or real. The higher order bits specify a page, and are therefore sometimes referred to as an "effective page number" or "real page number", as the case may be.

[0062] Computer system 100 contains an address translation mechanism for translating effective addresses generated by CPU 101 to real addresses in memory 102. This address translation mechanism includes a segment table mechanism 821 for mapping effective segment ID 811 to virtual segment ID 814, and a page table mechanism 822 for mapping virtual segment ID 814 and page number 812 to real page number 815. While these mechanisms are shown in FIG. 8 as single entities for illustrative purposes, they in fact comprise multiple tables or register at different levels. I.e., a complete page table and a complete segment table reside in main memory 102, while various smaller cached portions of the data in these tables is contained in CPU 101 itself or the L2 cache. There are additional translation mechanisms (not shown) which will in limited circumstances translate directly from an effective to a real address.

[0063] While CPU 101 supports address translation as illustrated in FIG. 8, it also supports more simple addressing. Specifically, one of the operating modes is a "tags active" mode, in which effective addresses are the same as virtual addresses (i.e., an effective segment ID 811 maps directly to virtual segment ID 814 without lookup, so that the high-order 16 bits of virtual segment ID are always 0). CPU 101 may also operate in an effective=real addressing mode.

[0064] Effective=real mode (E=R) is a special addressing mode, typically reserved for certain low level operating system functions which operate more efficiently if always stored at the same real address locations. These operating system functions may need to access reserved areas of memory, and therefore typically execute in a special privileged state (as opposed to most user executable code, which executes in a non-privileged state called a "problem state"). These operating system functions are created and tested by a process assumed to be trusted, in the sense that the resulting code will not cause unauthorized interference with machine processes. When executing in E=R mode and without logical partitioning, the lower order 40 bits of effective address (i.e., $EA_{24\ 63}$) generated by instruction unit **201** (in the case of instructions) or execution unit **211** (in the case of data) is the same as the real address ($RA_{24\ 63}$); the high order effective address bits are assumed to be 0. When operating in E=R mode, addresses are not translated, i.e., the page table mechanism and segment table mechanism, described above, along with any associated caches, are not used. This has the effect of mapping all E=R mode processes to the same real memory, even when executing on different processors. E=R mode addressing is active when either (a) an applicable address translate bit in one of the machine state registers is set off, or (b) under certain circumstances, when the effective address lies within a special reserved range of addresses. Appropriate hardware logic (not shown) detects these conditions and generates an E=R control signal for use by addressing logic.

different logical

(19)

5

(1)

[0065] In the preferred embodiment, computer system **100** can be logically partitioned. Logical partitioning means that the system is logically divided into multiple subsets called logical partitions, and some of the system resources are assigned to particular logical partitions, while other resources are shared among partitions. In the preferred embodiment, processors and real memory are assigned to logical partitions in a partitioned system, while buses, I/O controllers, and I/O devices are shared, it being understood that it would be possible to assign different types and mixtures of devices to partitions. In a logically partitioned system, each processor of the multiprocessor system is assigned to a partition, along with a subset of the real memory address space. With limited exceptions (explained below), tasks executing on a processor can only access real memory within that processor's subset of the real memory address space. This has the effect of isolating tasks executing on different processors in different logical partitions. From the standpoint of CPU and memory, the logically partitioned multiprocessor computer system behaves very much like multiple separate computer systems. This avoids some of the contention and other overhead issues associated with prior art multiprocessor systems. At the same time, the different logical partitions share hardware resources such as disk storage and I/O, as well as certain low level software resources. Thus, many of the advantages of a multiprocessor system over multiple discrete single processor systems are maintained. Furthermore, it is possible for multiple processors to share a single logical partition. For example, a computer system containing 16 processors could be configured in four logical partitions, each containing four processors, and resembling in certain characteristics the performance of four 4-way multiprocessor systems as opposed to a single 16-way multiprocessor system.

(1)

[0066] Since user executable (non-privileged) code is typically translated as described above from an effective address to a real address (with or without the intermediate virtual address), this same basic mechanism can be used to support logical partitioning. The operating system will assign a block of user-accessible address space to a block of real memory address space lying within the logical partition of the processor executing the user code. Subsequent references to an effective address within this block will be translated using the translation mechanisms to the corresponding block of real memory address space. Thus, user executable code will reference something within the logical partition of the processor, without affecting memory outside the processor's logical partition.

[0067] However, the translation mechanism can not enforce logical partitioning of address references in E=R mode. Generally, this is privileged code, created using a trusted process. Even though the code is created using a trusted process, there are performance reasons to isolate such code executing on different processors to different logical partitions. At the same time, there is still a need for some operating system functions to have access to the entire real memory.

[0068] To support logical partitioning, two privileged execution states are defined, in addition to the non-privileged "problem state". The privileged execution states are called "supervisor state" and "hypervisor state". Most privileged functions execute in the supervisor state, and are confined to the logical partition of the processor upon which they are executing. Supervisor state code may be untranslated, in which case the high-order effective address bits are directly manipulated by hardware to confine address references to the logical partition of the executing processor. In this manner, duplicates of these functions can concurrently execute on different processors in different logical partitions, without concern for the effect on other logical partitions. Only a select few functions, such as those which support logical partitioning itself, execute in the ultra-privileged hypervisor state, and have access to the full real address space of computer system **100**. Each executing thread has its own privilege state (either hypervisor, supervisor, or problem), which is independent of the privilege state associated with any other thread.

[0069] Processor state and configuration information is maintained in a set of special-purpose registers. **FIG. 9** illustrates some of these registers and associated control structures. The key register is Active-Thread Machine State Register (MSR) **901**, which maintains certain state information for the currently active thread. Dormant-Thread Machine State Register (MSRDorm) **902** maintains the same type of information for the currently dormant thread. Each register **901**, **902** contains the following respective bits, among others:

- **[0070]** DR bit, which indicates the corresponding thread's data addresses should be translated;
- **[0071]** IR bit, which indicates the corresponding thread's instruction addresses should be translated;
- **[0072]** Pr bit, which indicates whether the corresponding thread is in problem state;
- **[0073]** TA bit, which indicates "tags active" mode.
- **[0074]** HV bit, which indicates the corresponding thread is in hypervisor state;

[0075] **FIG. 9** illustrates respective data relocate (DR) signal lines **921**, **931**; instruction relocate (IR) signal lines **922**, **932**; problem state signal lines **923**, **933**; tags active signal lines **924**, **934**; and hypervisor state signal lines **925**, **935**.

[0076] A machine state register is not permanently associated with a thread; rather, there is one physical register **901** which always contains the information for the active thread, and another which contains the dormant thread's information. For this reason, an Active Thread Identifier bit **961** is needed to identify which is the active or dormant thread. ActThreadID bit **961** is kept in a separate special register. Upon a thread switch, the contents of registers **901** and **902** are swapped, and ActThreadID bit **961** is changed. Swapping register contents simplifies downstream control mechanisms, since in most cases only the contents of the active thread MSR **901** are relevant.

[0077] As shown in **FIG. 9**, input to each machine state register **901**, **902** is controlled by a respective multiplexer **903**, **904**, which receives inputs from various sources. The inputs to multiplexer **903** illustrate the various ways in which MSR **901** can be altered. Input path **941** represents a move to MSR (mtMSR) instruction, i.e., MSR **901** can be altered by executing a special mtMSR instruction while in a privileged state, which causes data to be loaded directly from a general purpose register into the MSR. Input path **942** represents an interrupt state, i.e., upon occurrence of an interrupt condition, the MSR is automatically loaded with a predefined state associated with the interrupt. Input paths **943** and **944** represent a System Call and a System Call Vectored, respectively. These are special processor instructions, typically made while in the problem state in order to invoke a privileged state. Both cause a predefined state to be loaded into MSR and a jump to a predefined location. System Call Vectored does not affect as many bits in the MSR as does System Call, i.e., System Call Vectored causes only a few bits to change, most of the bits being simply copied from their current state. Return from System Call Vectored (rfscv) path **945** represents reloading the MSR with its previous state upon return from a System Call Vectored; these values are stored in a special register (not shown). Return from interrupt/system call path **946** is conceptually similar to rfscv **945**, and represents reloading the MSR with its previous state upon return from an interrupt or a System Call. The previous state of the MSR is saved in SRR1 registers **905**, **906**, which are special purpose registers for holding a saved state. One SRR1 register is associated with each thread, and the state of MSR **901** is saved to the register associated with the currently active thread as identified by ActThreadID **961**. Upon return from an interrupt or System Call, multiplexer **907** selects the appropriate register **905** or **906** for restoring the previous MSR state. Input path **947** represents the contents of MSRDorm **902**, which is loaded into MSR **901** upon a thread switch. Input path **948** represents the current contents of MSR **901**; because some of the events which cause changes to MSR **901** do not affect all bits, this path represents a copying of non-affected bits back into MSR **901**.

[0078] MSRDorm **902** is altered in similar fashion, although fewer paths are shown in **FIG. 9** because an interrupt, System Call, or System Call Vectored can apply only to the currently active thread. Like MSR **901**, MSRDorm **902** can be altered by a special move to MSRDorm instruction, as represented by path **951**. MSRDorm will receive the contents of MSR **901** upon a thread switch, as represented by path **952**. Finally, path **953** represents copying bits not affected by a change back into MSRDorm **902**.

[0079] Also shown in **FIG. 9** is a set of configuration registers **910**. Unlike MSR **901** and MSRDorm **902**, these registers contain configuration information which is intended to change rarely, if at all. I.e., information in configuration registers **910** might be set upon initial installation of a system and might be altered upon major reconfiguration, such as the addition of processors to the system, or the system being re-partitioned. These registers can be altered only in hypervisor mode, i.e., are not intended to be written to from user executable code. Typically, information is loaded into configuration registers by a special-purpose service processor during system initialization. Among the information held in configuration registers **910** is a Logical Partitioning Environment Selector (LPES) bit **911** This bit is used to specify one of two operating system environments, designated "RS" and "AS". In the "RS" environment, non-hypervisor address references in E=R mode must be forced into the real memory subset of the processor's logical partition; in the "AS" environment, non-hypervisor address references in ER mode are not allowed. Configuration registers **910** also contain a 12-bit real memory offset field **912**, also referred to as a real memory offset register (RMOR), although it is physically part of the larger configuration register set **910**. Configuration registers **910** also contain a 2-bit real memory limit field **913**, also referred to as a real memory limit register (RMLR). Configuration registers **910** further contain a Logical Partition ID (LPID) field **914**, which is an identifier assigned to the logical partition to which the processor belongs.

[0080] The Pr bits **923**, **933** and HV bits **925**, **935** define the privilege state. If the HV bit is set, the corresponding thread is in the hypervisor state. If the HV bit is not set and the Pr bit is set, the corresponding thread is in the problem state. If nether bit is set, the corresponding thread is in the supervisor state. The HV bit can not be altered by a mtMSR instruction, for this would allow a thread in supervisor state to place itself in hypervisor state. The HV bit can only be set automatically by the hardware under certain predefined conditions, specifically certain interrupts (depending on the setting of LPES bit **911**) or certain System Calls, any of which cause instructions to branch to one of a set of predefined locations. Naturally, these predefined locations must contain trusted code suitable for execution in hypervisor state. All predefined locations associated with Hypervisor state are contained within a single real address subset at the low address range. This subset is reserved and can not be assigned to any processor of multiprocessor system **100**. The conditions for setting the HV bit can be summarized as follows:

$$\begin{gathered}
MSR(HV) \Leftarrow (\neg LPES \text{ AND (Any\_Interrupt OR System\_Call}_{26})) \text{ OR} \\
(LPES \text{ AND (Machine\_Check\_Interrupt OR} \\
\text{System\_Reset\_Interrupt OR System\_Call}_{26}))
\end{gathered}$$

[0081] Where System_Call$_{26}$ indicates a System Call (not including a System Call Vectored) in which bit **26** is set. Upon return from the interrupt or system call, the previous thread state is reloaded in the MSR register from one of SRR1 registers **905** or **906**. This previous state includes the previous value of HV bit **925**, and the HV bit is thus reset to its previous value

[0082] In a logically partitioned multiprocessor system, all address references in either problem or supervisor state should be confined to the logical partition associated with the processor which generated the address. Only in the hypervisor state should it be possible to reference an address outside this range.

[0083] **FIG. 7** illustrates at a conceptual level the generation of effective addresses of instructions in instruction unit **201**. Instruction unit **201** is capable of generating an address in any of a variety of ways. I0 Instruction Address Register **701** represents generation from an address in the instruction address register, i.e., an immediate address in the current thread. The most common way to generate an address is by incrementing this address, represented as path **711**. In some cases, the address in I0IAR **701** is used directly (e.g., when it was loaded into I0IAR **701**), represented as path **710**. Branch Address (relative) block **702** represents a relative branching instruction, in which an offset may be contained in the instruction or in a register. Because this is a branch relative instruction, the offset is added to low order address bits from I0IAR, and the high order bits from I0IAR may be incremented, decremented, or passed through unchanged. These bits are then combined, represented as path **712**. Bpipe-Base block **703** represents generation of an absolute branch through various means, usually using a hardware branch pipeline, such as branching to a value from a general purpose or a special register, a value derived as a combination of bits in the instruction and register bits, indirect addressing, etc. SCV block **704** represents an address resulting from a system call or interrupt condition, which branch to predefined locations. Fetch Instruction Address Register block **705** represents address generation for speculative conditions. As shown, this might involve generation of the next instruction address for a dormant thread (block **706**) or SRRO registers which hold return from interrupted addresses for the current thread (block **707**) and dormant thread (block **708**), these values being swapped on a thread switch.

[0084] Ideally, instruction unit **201** provides a constant stream of instructions for decoding in decoder **206**, and execution by execution unit **211**. L11-cache **106** must respond to an access request with minimal delay. Where a requested instruction is actually in L11-cache, it must be possible to respond and fill the appropriate buffer without requiring decoder/dispatcher **206** to wait. Where L11-cache can not respond (i.e., the requested instruction is not in L11-cache), a longer path via cache fill bus **233** through memory management unit **222** must be taken. In this case, the instruction may be obtained from L2 cache **108**, from main memory **102**, or potentially from disk or other storage. It is also possible that the instruction will be obtained from L2 cache of another processor. In all of these cases, the delay required to fetch the instruction from a remote location may cause instruction unit **201** to switch threads. I.e., the active thread becomes inactive, the previously inactive thread becomes active, and the instruction unit **201** begins processing instructions of the previously inactive thread held in thread switch buffer **204**.

[0085] **FIG. 3** illustrates the major components of L11-cache **106** in greater detail than shown in **FIG. 1** or **2**, according to the preferred embodiment. L11-cache **106** includes effective-to-real address table (ERAT) **301**, I-cache directory array **302**, and I-cache instruction array **303**. I-cache instruction array **303** stores the actual instructions which are supplied to instruction unit **201** for execution. I-cache directory array **302** contains a collection of real page numbers, validity bits, and other information, used to manage instruction array **303**, and in particular to determine whether a desired instruction is in fact in the instruction array **303**. ERAT **301** contains pairs of effective page numbers and real page numbers, and is used for associating effective with real addresses.

[0086] When instruction unit **201** requests an instruction from I-cache **106**, providing an effective address of the requested instruction, I-cache must rapidly determine whether the requested instruction is in fact in the cache, return the instruction if it is, and initiate action to obtain the instruction from elsewhere (e.g., L2 cache, main memory) if it is not. In the normal case where the instruction is in fact in L11-cache **106**, the following actions occur concurrently within the I-cache, as illustrated in **FIG. 3**:

[0087] (a) The effective address from instruction unit **201** is used to access an entry in ERAT **301** to derive an effective page number and associated real page number.

[0088] (b) The effective address from instruction unit **201** is used to access an entry in directory array **302** to derive a pair of real page numbers.

[0089] (c) The effective address from instruction unit **201** is used to access an entry in instruction array **303** to derive a pair of cache lines containing instructions.

[0090] In each case above, the input to any one of ERAT **301**, directory array **302**, or instruction array **303**, is not dependent on the output of any other one of these components, so that none of the above actions need await completion of any other before beginning. The output of the ERAT **301**, directory array **302**, and instruction array **303** are then processed as follows:

[0091] (a) The effective page number from ERAT **301** is compared with the same address bits of the effective address from instruction unit **201** in comparator **304**; if they match, there has been an ERAT "hit". (But where addressing in E=R mode, the ERAT is always deemed "hit" regardless of the comparison, as explained below.)

[0092] (b) The real page number from ERAT **301** is compared with each of the real page numbers from directory array **302** in comparators **305** and **306**; if either of these match, and if there has been an ERAT hit, then there is an I-cache "hit", i.e., the requested instruction is in fact in I-cache **106**, and specifically, in instruction array **303**.

[0093] (c) The output of the comparison of real page numbers from ERAT **301** and directory array **302** is used to select (using selection multiplexer **307**) which of the pair of cache lines from instruction array **303** contains the desired instruction.

[0094] Performing these actions concurrently minimizes delay where the desired instruction is actually in the I-cache. Whether or not the desired instruction is in the I-cache, some data will be presented on the I-cache output to instruction unit **201**. A separate I-cache hit signal will indicate to instruction unit **201** that the output data is in fact the desired instruction; where the I-cache hit signal absent, instruction unit **201** will ignore the output data. The actions taken by I-cache **106** in the event of a cache miss are discussed later herein.

[0095] **FIG. 4** shows in greater detail ERAT **301**, and associated control structures. ERAT **301** is an 82-bitx128 array (i.e, contains 128 entries, each having 82 bits). Each ERAT entry contains a portion (bits **0-46**) of an effective address, a portion (bits **24-51**) of a real address, and several additional bits described below. ERAT **301** may be thought of as a small cache directly mapping a subset of effective addresses to their respective real addresses, thus avoiding the delays inherent in the address translation mechanism depicted in **FIG. 8**, described above. Because ERAT **301** is a cache of the larger mapping structures, mapped-to real addresses within the ERAT are confined to the logical partition of the processor which generated the effective address if partition integrity is maintained within the larger mapping structures, which is the responsibility of the operating system.

[0096] ERAT **301** is accessed by constructing a hash function of bits **45-51** of the effective address (EA), along with two control lines: multi-thread control line (MT), which indicates whether multithreading is active (in the CPU design of the preferred embodiment, it is possible to turn multithreading off); and ActThreadID line **961**. The hash function (HASH) is as follows:

$$HASH_{06} = (EA_{45}\ \mathrm{AND}\neg\, MT)\ \mathrm{OR}\ (ActThreadID\ \mathrm{AND}\ MT) \| EA_{46} \|$$

$$EA_{38}\, XOR\ EA_{47} \| EA_{39}\, XOR\ EA_{48} \| EA_{49\,51}$$

[0097] As can be seen, this is a 7-bit function, which is sufficient to specify any one of the 128 entries in the ERAT. Select logic **401** selects the appropriate ERAT entry in accordance with the above hash function.

[0098] Comparator **304** compares bits **0-46** of the effective address generated by instruction unit **201** with the effective address portion of the selected ERAT entry. Because bits **47-51** of the effective address from instruction unit **201** were used to construct the hash function, it can be shown that a match of bits **0-46** is sufficient to guarantee a match of the full effective page number portion of the address, i.e. bits **0-51**. A match of these two address portions means that the real page number ($RA_{24\ 51}$) in the ERAT entry is in fact the real page number corresponding to the effective address page number ($EA_{0\ 51}$) specified by instruction unit **201**. For this reason, the effective address portion stored in an ERAT entry is sometimes loosely referred to as an effective page number, although in the preferred embodiment it contains only bits **0-46** of the effective page number.

[0099] Because the ERAT effectively by-passes the address translation mechanisms described above and depicted in **FIG. 8**, the ERAT duplicates some of the access control information contained in the normal address translation mechanism. I.e., a translation of effective address to real address will normally verify access rights through additional information contained in segment table **821**, page table **822**, or elsewhere. ERAT **301** caches a subset of this information to avoid the need to refer to these address translation mechanisms. Further information about the operation of the ERAT can be found in U.S. Pat. No. 6,021,481, entitled Effective-To-Real Address Cache Managing Apparatus and Method, herein incorporated by reference.

[0100] Each ERAT entry contains several parity, protection, and access control bits. In particular, each ERAT entry includes a cache inhibit bit, a problem state bit, and an access control bit. Additionally, separate array **403** (1 bitx 128) contains a single valid bit associated with each respective ERAT entry. Finally, a pair of tag mode bits is stored in separate register **404**. The valid bit from array **403** records whether the corresponding ERAT entry is valid; a variety of conditions might cause processor logic (not shown) to reset the valid bit, causing a subsequent access to the corresponding ERAT entry to reload the entry. The cache inhibit bit is used to inhibit writing the requested instruction to I-cache instruction array **303**. I.e., although a range of addresses may contain an entry in ERAT, it may be desirable to avoid caching instructions in this address range in the I-cache. In this case, every request for an instruction in this address range will cause the line fill sequence logic (described below) to obtain the requested instruction, but the instruction will not be written to array **303** (nor will directory array **302** be updated). The problem state bit records the "problem state" of the active thread (from MSR(Pr) bit **923**) at the time the ERAT entry is loaded. A thread executing in privileged state generally has greater access rights than one in problem state. If an ERAT entry were loaded during one state, and the problem state subsequently changed, there is a risk that the currently executing thread should not have access to addresses in the range of the ERAT entry, and this information must accordingly be verified when the ERAT is accessed. The access control bit also records access information at the time the ERAT entry was loaded, and is checked at the time of access. Tag mode bits **404** record the tag mode of the processor (tags active or tags inactive) when the ERAT was loaded; there is one tag mode bit associated with each half (64 entries) of the ERAT, which is selected using the 0 bit of the ERAT HASH function. Since tag mode affects how effective addresses are interpreted, a change to tag mode means that the real page numbers in the ERAT entry can not be considered reliable. It is expected that the tag mode will change infrequently, if ever. Therefore, if a change is detected, all entries in the corresponding half of the ERAT are marked invalid, and are eventually reloaded.

[0101] When CPU **101** is executing in effective=real mode, the ERAT is effectively bypassed. In a non-logically partitioned system, E=R would imply that the lower order 40 bits of effective address (i.e., $EA_{24\ 63}$) generated by instruction unit **201** are the same as the real address ($RA_{24\ 63}$), and hence any real address is potentially accessible. Logical partitioning requires that the effective addresses (base real address) be converted to a partitioned real address, i.e. one that is confined to some subset of the real address space. Bitwise OR logic **422** performs a logical OR of each bit in real memory offset register (RMOR) from configuration register set **910**, with a corresponding bit of effective address in the range of bits **24** to **35**, i.e., 12 bits in all are ORed. The bits in the RMOR correspond to the real address space of a logical partition. When using E=R mode and not in hypervisor state, the high order effective address bits in the range of those which enforce logical partitioning should all be zeroes. OR logic **422** is used instead of simple concatenation in order to support logically partitioned real address space subsets of different sizes. In the preferred embodiment, real address space subset sizes of 64 GB (236 bytes), 4 GB (232 bytes) and 256 MB (2^{28} bytes) are supported. For example, when a partition size of 64 GB is being used, the 4 high order bits in RMOR will identify a real address space subset allocated to a logical partition, the 8 low order bits of RMOR must be set to 0, $EA_{24\ 27}$ must be 0, and $EA_{28\ 63}$ will specify a real address within the subset of the logical partition. Similarly, where a real address space subset size of 256 MB is being used, all 12 bits of the RMOR will identify a real address space subset, $EA_{24\ 35}$ must be 0, and $EA_{36\ 63}$ will specify a real address within the logical partition. In hypervisor state, a processor has access to the entire real memory address space and system resources, and the RMOR is therefore by-passed. Additionally, the RMOR is by-passed when LPES bit **911** is 0, indicating that computer system **100** is configured in "AS" environment. As shown in **FIG. 4**, HV bit **620** and LPES bit **911** control multiplexer **421**, which selects effective address bits **24-35** ($EA_{24\ 35}$) if either of these conditions is present, and otherwise selects the output of OR logic **422**.

[0102] As shown in **FIG. 4**, when control line E=R is active, selection multiplexer **402** selects $RA_{24\ 51}$ from the selected ERAT entry as the real page number (RPN) output when E=R is false, and multiplexer **402** selects the output of multiplexer **421**, concatenated with $EA_{36\ 51}$ when E=R is true. Additionally, where E=R is true, the ERAT is deemed to be hit regardless of the comparison result in comparator **304**.

[0103] ERAT logic **405** generates several control signals which control the use of the RPN output of selection multiplexer **402** and ERAT maintenance, based on the output of selector **304**, the effective=real mode, the various bits described above, and certain bits in the CPU's Machine State Register (or MSRDorm, as the case may be). In particular, logic **405** generates ERAT Hit signal **410**, Protection Exception signal **411**, ERAT Miss signal **412**, and Cache Inhibit signal **413**.

[0104] ERAT Hit signal **410** signifies that the RPN output of selection multiplexer **402** may be used as the true real page number corresponding to the requested effective address. This signal is active when effective=real (by-passing the ERAT); or when comparator **304** detects a match and there is no protection exception and certain conditions which force an ERAT miss are not present. This can be expressed logically as follows:

$$\text{ERAT\_Hit} = (E = R)\ \text{OR}\ (\text{Match\_304 AND} \neg\ \text{Protection\_Exc AND}$$
$$\neg\ \text{Force\_Miss})$$

[0105] Where Match_**304** is the signal from comparator **304** indicating that $EA_{0\ 46}$ from instruction unit **201** matches $EA_{0\ 46}$ in the ERAT entry.

[0106] Protection Exception signal **411** signifies that, while the ERAT entry contains valid data, the currently executing process is not allowed to access it. ERAT Miss signal **412** indicates that the requested ERAT entry does not contain the desired real page number, or that the entry can not be considered reliable; in either case, the ERAT entry must be reloaded. Cache inhibit signal **413** prevents the requested instruction from being cached in instruction array **303**. These signals are logically derived as follows:

$$\text{Force\_Miss} = \neg\ \text{Valid OR}\ (MSR(Pr) \neq ERAT(Pr))\ \text{OR}\ (MSR(TA)$$
$$\neq \text{Tag\_404})$$
$$\text{Protection\_Exc} = \neg\ (E = R)\ \text{AND} \neg\ \text{Force\_Miss AND Match\_304 AND}$$
$$ERAT(\text{AC})\ \text{AND}\ (MSR(Us)\ \text{OR} \neg\ MSR(TA))$$
$$\text{ERAT\_Miss} = \neg\ (E = R)\ \text{AND}\ (\neg\ \text{Match\_304 OR Force\_Miss})$$
$$\text{Cache\_Inhibit} = \neg\ (E = R)\ \text{AND}\ ERAT(CI)$$

[0107] Where:

[0108] Valid is the value of valid bit from array **403**;

[0109] ERAT(Pr) is the problem state bit from the ERAT entry;

[0110] ERAT(AC) is the access control bit from the ERAT entry;

[0111] ERAT(CI) is the cache inhibit bit from the ERAT entry;

[0112] MSR(TA) is the tags active bit from the Machine State Register;

[0113] MSR(Us) is the User state bit from the Machine State Register; and

[0114] Tag_**404** is the selected tag bit from register **404**.

[0115] I-cache directory array **302** and contains **512** entries, each having a pair of real page numbers, validity bits, parity bits, and a most-recently-used bit. An entry in array **302** is selected using effective address bits **48-56** ($EA_{48\ 56}$), which are used as a sparse hash function. Because there is no guarantee that either of the real page numbers contained in an entry in array **302** correspond to the full effective address page number of the desired instruction, both selected real page numbers are simultaneously compared with the real page number output 411 of ERAT **301**, using comparators **305** and **306**. The output of these and certain other logic determines which real page number, if any can be used. $EA_{48\ 58}$ simultaneously selects an entry from instruction array **303**, and the results of comparators **305**, **306** are used to select which set (i.e., which half of the entry) contains the associated instruction.

[0116] The above text describes the situation where the instruction sought is actually in the I-cache. Where there has been an I-cache miss, there are two possibilities: (a) there has been an ERAT hit, but the instruction is not in the instruction array; or (b) there has been an ERAT miss. In the case where there has been an ERAT hit, it is possible to fill the desired cache line significantly faster. Because the real page number is in the ERAT, the desired data is known to be in main memory (and possibly in an L2 cache). It is possible for logic in L11-cache **106** to construct the full real address of the desired instruction from ERAT data, without accessing external address translation mechanisms, and to fetch this data directly from L2 cache or memory. In the case where there has been an ERAT miss, an external address translation mechanism must be accessed in order to construct the real address of the desired instruction, and to update the ERAT as necessary with the new real page number. It is possible that in this case, the desired data will not exist in main memory at all, and will have to be read in from secondary storage such as a disk drive.

[0117] Further information concerning the operation of L-11-cache **106** is contained in U.S. Pat. No. 6,161,166, entitled Instruction Cache for Multithreaded Processor, herein incorporated by reference.

[0118] As described above, OR logic **422** performs a logical OR of address bits from the RMOR and the effective address to create an logically partitioned effective address which is offset from the effective address generated by instruction unit **201**. The use of OR logic presumes that certain high order bits of the effective address are zeroes, otherwise the bits identifying the logical partition can be corrupted. These conditions and others are verified by address protection logic shown in **FIG. 6**.

[0119] As shown in **FIG. 6**, the 2-bit real memory limit register (RMLR) **913** and effective address bits **24-35** (EA$_{24}$ $_{35}$) are input to partition size decode logic **601**. The RMLR designates the size of the logical partitions, as follows:

RMLR value:	0 0	Partition size:	64 GB
	1 0		4 GB
	1 1		256 MB

[0120] Decode logic **601** outputs an address-out-of-range signal **604**, a single bit value which is a logic '1' if the effective address runs outside the established partition size as specified in the RMLR. The logic function performed by decode logic **601** can be expressed as:

$$\begin{aligned}AOR = {} & EA_{24}\text{ OR }EA_{25}\text{ OR }EA_{26}\text{ OR }EA_{27}\text{ OR }(RMLR_0\text{ AND }EA_{28})\text{ OR}\\ & (RMLR_0\text{ AND }EA_{29})\text{ OR }(RMLR_0\text{ AND }EA_{30})\text{ OR}\\ & (RMLR_0\text{ AND }EA_{31})\text{ OR }(RMLR_1\text{ AND }EA_{32})\text{ OR}\\ & (RMLR_1\text{ AND }EA_{33})\text{ OR }(RMLR_1\text{ AND }EA_{34})\text{ OR}\\ & (RMLR_1\text{ AND }EA_{35})\end{aligned}$$

[0121] Decode logic **601** generates an AOR signal as described above for all effective addresses generated by instruction unit **201**. However, the signal is significant only if certain conditions are met. Specifically, if the effective address is translated through the address translation mechanism shown in **FIG. 8**, then the AOR signal is ignored because there is no correspondence of high order effective address bits and real address bits, and logical partitioning code under the control of the operating system assures that values in the translation tables enforce logical partitioning. The AOR signal is also ignored if the thread for which the address is generated is in hypervisor state, since such a thread is authorized to access all logical partitions. Finally, the AOR signal is ignored if the LPES bit is 0 (indicating an "AS" system environment).

[0122] The logic which performs these functions is shown in **FIG. 6** as selectors **610**, **611**, and RS real address space check logic **602**. Selector **610** selects either MSR(IR) signal **922** or MSRDorm(IR) signal **932**, depending on incoming signal from DTA (Dormant Thread Address access select) line **612**. DTA line **612** is active when the effective address is generated on behalf of the dormant thread, i.e., in the case of a background fetch of the dormant thread's instructions. In all other cases, the DTA line is low, indicating that the address is generated on behalf of the active thread. Selector **610** outputs on IR line **621** the MSRDorm(IR) signal if DTA line **612** is active, otherwise outputs the MSR(IR) signal. Selector **611** similarly selects either MSR(HV) signal **925** or MSRDorm(HV) signal **935**, depending on DTA input **612**. The output of selector **611**, designated HV **620**, is also used as input to multiplexer **421**. The outputs of selectors **610** and **611** can be logically expressed as follows:

[0123] IR=(¬DTA AND MSR(IR)) OR (DTA AND MSRDorm(IR))

[0124] HV=(¬DTAAND MSR(HV)) OR (DTAAND MSRDorm(HV))

[0125] The output of RS real address space check logic **602** can be expressed as follows:

[0126] RS_check=LPES AND AOR AND ¬HV AND ¬IR

[0127] Where an "AS" mode operating system is used, AS real address space check logic **603** will generate an AS check signal if there is an attempt to generate an address in E=R mode, while not in hypervisor state. In other words, when in "AS" mode, E=R addressing can only be used in hypervisor state. The output of AS real address space check logic **603** can be expressed as follows:

[0128] AS_check=¬LPES AND ¬HV AND ¬IR

[0129] As shown in **FIG. 6**, an instruction storage interrupt is generated if there is an AS check or if there is and RS check, i.e.

[0130] LPAR ISI=AS_check OR RS_check

[0131] This is simply one set of possible conditions which may cause an interrupt. A protection exception signal **411** (explained above) also causes an instruction storage interrupt, as do various other conditions. The effect of the instruction storage interrupt is that the generated address is not accessed by the processor, and appropriate interrupt routines are called.

[0132] The above text and accompanying figures explain how addresses of instructions are verified and mapped to an address range corresponding to the logical partition of the processor which generated the address. Addresses of data are processed in a similar, although simplified, manner. Data addresses are processed using the logic depicted in FIGS. 5 and 6. Much of the logic which processes data addresses is physically separate from the logic which processes instruction addresses, although the two operate in a similar manner.

[0133] Unlike instructions (which may be pre-fetched for either the active or dormant thread), only the active thread generates data addresses. Therefore some of the logic shown in **FIG. 6**, which is required to process pre-fetched instruction addresses for a dormant thread, is not needed in the case of data addresses. Additionally, the L1 data cache does not use an ERAT.

[0134] FIG. 5 depicts the real address partitioning mechanism for data addresses, analogous to the partitioning mechanism for instruction addresses shown in FIG. 4. Execution unit 211 generates a data address by any of various conventional means known in the art, e.g., as a value taken from a register, as a field of an instruction concatenated or offset from a register value, as a computed value from multiple registers, etc. The effective address may or may not require translation. Where translation is indicated (E=R is false), address bits 0-51 are input to the translation mechanism (depicted at a high level in FIG. 8), which produces a translated 28-bit real page number. Where translation is not indicated (E=R is true), a partitioned real address is produced from the effective address in a manner similar to that explained above for instruction addresses. I.e., $EA_{24\ 35}$ is bitwise ORed with the contents of real memory offset register 912 by OR logic 522. Multiplexer 521 selects $EA_{24\ 35}$ if either HV 620 or LPES 911 is true, otherwise selects the output of logic 522. The output of multiplexer 521 is concatenated with $EA_{36\ 51}$ for input to multiplexer 502. Multiplexer 502 chooses either translated 28-bit real page number from the translation mechanism, or the output of multiplexer 521, as the 28-bit real page number, i.e., real address bits 24 to 51. The 12-bit byte index within the real page is taken directly from $EA_{52\ 63}$.

[0135] As in the case of instruction addresses, separate logic circuitry for data addresses produces an error signal. This logic is similar to that shown in FIG. 6, but simplified. In particular, because data addresses are only generated on behalf of the currently active thread, selectors 610 and 611 are not used in the logic which checks for LPAR address errors in data addresses. I.e., in the case of data addresses, HV=MSR(HV) and DR=MSR(DR). AS Real Address Space Check logic 603 is similarly simplified because only MSR(TA) and MSR(Pr) (and not MSRDorm(TA) and MSR-Dorm(Pr)) are used as input.

[0136] LPID 914 is used as a tag in certain bus operations to identify the relevant logical partition, thus limiting the effect of the bus operation and improving efficiency. A processor receiving data in such an operation from a bus to which it is attached will compare the tag received on the bus (the logical partition ID to which the operation pertains) with its own logical partition ID stored in its configuration register 910. If the two are not identical, the operation is ignored by the processor.

[0137] A simple example will demonstrate the potential performance improvement of this arrangement. ERAT 301 is essentially a cache of some of the information contained in segment table 821 and page table 822, the segment and page tables being external to the processor. Each logical partition has its own segment and page tables, which are maintained independently of those in other logical partitions. Since a logical partition may contain multiple processors, activity in another processor may cause a page fault or other condition which alters the contents of one or the other of these tables. In that event, the corresponding ERAT entries may be affected. Therefore, whenever the segment table or page table are modified, an appropriate message will be broadcast to all processors on the bus, so that each may invalidate any affected ERAT entry. If, however, a processor is in a different logical partition, its ERAT is not affected by such a change. By comparing the LPID in the bus tag with the processor's own LPID in its configuration register, the processor knows immediately (e.g., at the bus interface 225, without accessing ERAT 301) whether the bus message pertains to it, and can safely ignore any page table or segment table changes in for different logical partition.

[0138] The ability of code in hypervisor state to alter the information in configuration register 910 means that the logical partitioning of a system can be dynamically changed. E.g., processors and other resources can be re-allocated to different logical partitions, the address ranges associated with a logical partition can be altered, or partitioning can be turned off entirely. Since only code executing in hypervisor state can alter these registers, the system is protected from accidental re-configuration by user code.

[0139] Additional background information concerning an exemplary (although by no means the only possible) hypervisor implementation can be found in commonly assigned copending U.S. patent application Ser. No. 09/314,214, filed May 19, 1999, entitled Logical Partition Manager and Method, herein incorporated by reference.

[0140] It will be understood that certain logic circuitry not essential to an understanding of the present invention has been omitted from the drawings and description herein for clarity. For example, logic for maintaining the MRU bit in array 302, logic for detecting parity errors and taking appropriate corrective action, etc., have been omitted.

[0141] In the preferred embodiment, a multithreaded processor employing coarse-grained hardware multithreading concepts is used. However, it will be understood that as alternative embodiments it would be possible to employ fine-grained multithreading operation, in which execution among the various threads is rotated on a cycle-by-cycle basis. It would also be possible to support logical partitioning as described herein on a processor which does not have hardware multithreading support.

[0142] While the invention has been described in connection with what is currently considered the most practical and preferred embodiments, it is to be understood that the invention is not limited to the disclosed embodiments, but on the contrary, is intended to cover various modifications and equivalent arrangements included within the spirit and scope of the appended claims.

What is claimed is:

1. A computer processing apparatus, comprising:

effective address generating logic, said effective address generating logic generating effective addresses to be accessed, wherein at least some of said effective addresses are real addresses;

at least one state register for recording processor operating parameters, said at least one state register including a mode designator, said mode designator designating an operating mode;

real address range checking logic, said real address range checking logic generating an error signal responsive to detecting a real address outside a predetermined range when said real address is associated with a first operating mode, and not generating an error signal responsive to detecting a real address outside said predetermined range when said real address is associated with a second operating mode.

2. The computer processing apparatus of claim 1, further comprising:

a plurality of sets of registers for supporting the execution of a plurality of threads, each set of registers corresponding to a respective one of said plurality of threads;

wherein said at least one state register for recording processor operating parameters comprises a respective mode designator associated with each thread, said mode designator designating, for each of said threads independently, a respective operating mode; and

wherein said real address range checking logic generates an error signal responsive to detecting a real address outside said predetermined range when said real address is generated on behalf of a thread in said first operating mode, and does not generate an error signal responsive to detecting a real address outside said predetermined range when said real address is generated on behalf of a thread in said second operating mode.

3. The computer processing apparatus of claim 2, wherein said effective address generating logic comprises an instruction unit for generating effective addresses of instructions for execution by said processor, said instruction unit generating effective addresses on behalf of an active thread and on behalf of at least one dormant thread, said real address range checking logic selectively generating an error signal responsive to detecting a real address outside said predetermined range on behalf of an active thread and on behalf of at least one dormant thread responsive to said mode designator associated with the respective thread.

4. The computer processing apparatus of claim 1,

wherein at least some of said effective addresses generated by said effective address generation logic are translatable addresses intended for translation from an effective address to a real address; and

wherein said computer processing apparatus further comprises real address selection logic for selectively outputting a real address translated from said effective address when said effective address is a translatable address, and outputting a real address generated by said real address partitioning logic when said effective address is a base real address.

5. The computer processing apparatus of claim 4, further comprising:

an effective-to-real address translation table, wherein at least some of said effective addresses which are translatable addresses are translated from an effective address to a real address by reference to said effective-to-real address translation table.

6. The computer processing apparatus of claim 1, wherein said real address range checking logic further comprises:

a real memory limit register specifying a range of address bits of said effective address;

AND logic performing a plurality of logical ANDs of each of a plurality of address bits derived from said effective address generated by effective address generation logic with a respective bit from a mask generated from a value in said real memory limit register to produce a masked portion of said effective address; and

OR logic for performing a logical OR of a plurality of address bits derived from said effective address generated by said effective address generation logic, said plurality of address bits derived from said effective address including said masked portion of said effective address generated by said effective address generation logic.

7. The computer processing apparatus of claim 1, wherein said mode designator is placed in said second operating mode only upon occurrence of one of a set of predefined events.

8. The computer processing apparatus of claim 7, wherein each said predefined event of said set of predefined events causes said computer processing apparatus to branch to a respected predefined real memory address.

9. The computer processing apparatus of claim 7, wherein a state represented by said at least one state register is saved in a saved state register upon occurrence of one of said set of predefined events, and restored to said at least one state register upon return from processing said one of said set of predefined events.

10. A computer system, comprising:

a plurality of processors;

a main memory addressable using a real address;

a logical partitioning mechanism capable of partitioning said computer system into a plurality of logical partitions, each processor of said plurality of processors being assigned by said logical partitioning mechanism to a respective one of said logical partitions, each logical partition being assigned a corresponding respective discrete range of real addresses of said main memory;

wherein each of said plurality of processors comprises:

a mode designator, said mode designator designating an operating mode for a thread executing in said processor; and

real address range checking logic, said real address range checking logic generating an error signal responsive to detecting a real address outside the corresponding respective discrete range of real addresses assigned to the logical partition to which the respective processor is assigned when said real address is generated on behalf of a thread executing in a first operating mode, and not generating an error signal responsive to detecting a real address outside said corresponding respective discrete range of real addresses assigned to the logical partition to which the respective processor is assigned when said real address is generated on behalf of a thread executing in a second operating mode.

11. The computer system claim 10, wherein each of said plurality of processors supports the execution of a plurality of threads and further comprises:

a plurality of mode designators, each mode designator corresponding to a respective one of said plurality of threads and designating, for each of said threads independently, a respective operating mode;

wherein said real address range checking logic generates an error signal responsive to detecting a real address outside said corresponding respective discrete range of real addresses assigned to the logical partition to which the respective processor is assigned when said real address is generated on behalf of a thread in said first operating mode, and does not generate an error signal responsive to detecting a real address outside said corresponding respective discrete range of real addresses assigned to the logical partition to which the respective processor is assigned when said real address is generated on behalf of a thread in said second operating mode.

12. The computer system of claim 10, wherein said real address range checking logic in each said processor further comprises:

a real memory limit register specifying a range of address bits of an effective address generated by effective address generation logic in said processor;

AND logic performing a plurality of logical ANDs of each of a plurality of address bits derived from said effective address generated by said effective address generation logic with a respective bit from a mask generated from a value in said real memory limit register to produce a masked portion of said effective address; and

OR logic for performing a logical OR of a plurality of address bits derived from said effective address generated by said effective address generation logic, said plurality of address bits derived from said effective address including said masked portion of said effective address generated by said effective address generation logic.

13. The computer system of claim 10,

wherein each of said plurality of processors generates effective addresses on behalf of threads executing in the processor, wherein at least some of said effective addresses are translatable addresses intended for translation from an effective address to a real address; and

wherein each of said plurality of processors further comprises real address selection logic for selectively outputting a real address translated from said effective address when said effective address is a translatable address, and outputting a real address generated by said real address partitioning logic when said effective address is a base real address.

14. The computer system of claim 13,

wherein each of said plurality of processors further comprises an effective-to-real address translation table, wherein at least some of said effective addresses which are translatable addresses are translated from an effective address to a real address by reference to said effective-to-real address translation table.

15. A computer processing apparatus for supporting a computer system divisible into a plurality of logical partitions, each partition having a respective portion of the real address space of said computer system, said computer processing apparatus comprising:

means for maintaining state information for at least one thread, said state information including an operating mode;

means for generating effective addresses for said at least one thread, wherein at least some of said effective addresses are base real addresses not intended for translation; and

means for detecting a real address outside a predetermined range and generating an error signal responsive thereto when said real address is generated for a thread executing in a first operating mode, and not generating an error signal when said real address is generated for a thread executing in a second operating mode.

16. The computer processing apparatus of claim 14,

wherein said means for maintaining state information maintains state information for each of a plurality of threads independently; and

wherein said means for detecting a real address outside a predetermined range generates an error signal responsive to state information for the thread for which the real address is generated.

17. A computer processing apparatus, comprising:

a configuration register for recording configuration information, said configuration information including a processor logical partition identifier;

at least one state register for recording processor operating parameters, said at least one state register including a mode designator, said mode designator designating an operating mode;

execution logic for executing instructions, said instructions including at least one instruction for altering said processor logical partition identifier, wherein said execution logic executes said at least one instruction for altering said processor logical partition identifier when in a first operating mode, and does not execute said at least one instruction for altering said processor logical partition identifier when in a second operating mode; and

bus interface logic, said bus interface logic receiving communications on a bus, at least some of said communications including a respective bus logical partition identifier;

wherein said processing apparatus ignores a communication including a bus logical partition identifier if said bus logical partition identifier does not match said processor logical partition identifier.

18. The computer processing apparatus of claim 15, further comprising:

a plurality of sets of registers for supporting the execution of a plurality of threads, each set of registers corresponding to a respective one of said plurality of threads;

wherein said at least one state register for recording processor operating parameters comprises a respective mode designator associated with each thread, said mode designator designating, for each of said threads independently, a respective operating mode; and

wherein said execution logic executes said at least one instruction for altering said processor logical partition identifier when said at least one instruction for altering said processor logical partition identifier is part of a thread executing in said first operating mode, and does not execute said at least one instruction for altering said processor logical partition identifier when said at least one instruction for altering said processor logical partition identifier is part of a thread executing in said second operating mode.

* * * * *

US006510496B1

(12) **United States Patent**
Tarui et al.

(10) **Patent No.: US 6,510,496 B1**
(45) **Date of Patent: Jan. 21, 2003**

(54) SHARED MEMORY MULTIPROCESSOR SYSTEM AND METHOD WITH ADDRESS TRANSLATION BETWEEN PARTITIONS AND RESETTING OF NODES INCLUDED IN OTHER PARTITIONS

(75) Inventors: **Toshiaki Tarui**, Sagamihara (JP); **Toshio Okochi**, Cambridge (GB); **Shinichi Kawamoto**, Hachioji (JP)

(73) Assignee: **Hitachi, Ltd.**, Tokyo (JP)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/501,978**

(22) Filed: **Feb. 11, 2000**

(30) **Foreign Application Priority Data**

Feb. 16, 1999 (JP) 11-036811

(51) **Int. Cl.**^{7} **G06F 12/10**; G06F 13/00

(52) **U.S. Cl.** **711/147**; 711/153; 711/202; 711/173; 711/146; 711/209; 709/215

(58) **Field of Search** 711/147, 153, 711/202, 173, 141, 170, 206, 146, 209; 709/213, 215

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,843,541	A	*	6/1989	Bean et al.	710/36
5,862,357	A	*	1/1999	Hagersten et al.	710/305
5,898,883	A	*	4/1999	Fujii et al.	712/28
5,923,847	A	*	7/1999	Hagersten et al.	709/215
5,940,870	A	*	8/1999	Chi et al.	711/206
6,088,770	A	*	7/2000	Tarui et al.	711/148
6,295,584	B1	*	9/2001	DeSota et al.	711/147
6,334,177	B1	*	12/2001	Baumgartner et al.	712/13

FOREIGN PATENT DOCUMENTS

JP	10-240707	*	9/1998

OTHER PUBLICATIONS

"Hive: Fault Containment for Shared Memory Multiprocessors," 15th ACM Symposium on Operating Systems Principles, Dec. 3–6, 1995, Copper Mountain Resort, Colorado, pp. 12–25.*

"Gigaplane–XB: Extending the Ultra Enterprise Family," HOT Interconnects V, Aug. 1997, pp. 97–112.*

* cited by examiner

Primary Examiner—Glenn Gossage

(74) *Attorney, Agent, or Firm*—Antonelli, Terry, Stout & Kraus, LLP

(57) ABSTRACT

A symmetric multiprocessor (SMP) of hierarchical connection realizing an inter-partition shared memory has at the gateway of an inter-node connection switch from each node, a translator for translating an address of an access command for an area shared between partitions, between a real address used in a partition and a shared area address used in common between partitions. Thereby, the address of a local area of each partition is freely set, and cache coherent control of a shared area is conducted at high speed by using a snoop command of the hierarchical connection SMP. Fault containment between partitions is realized by checking conformity between the address of the access command issued from another partition and the shared area configuration. Nodes included in other partitions may be reset from each partition. In addition, the configuration information of the shared area between partitions may be dynamically modified.

17 Claims, 16 Drawing Sheets

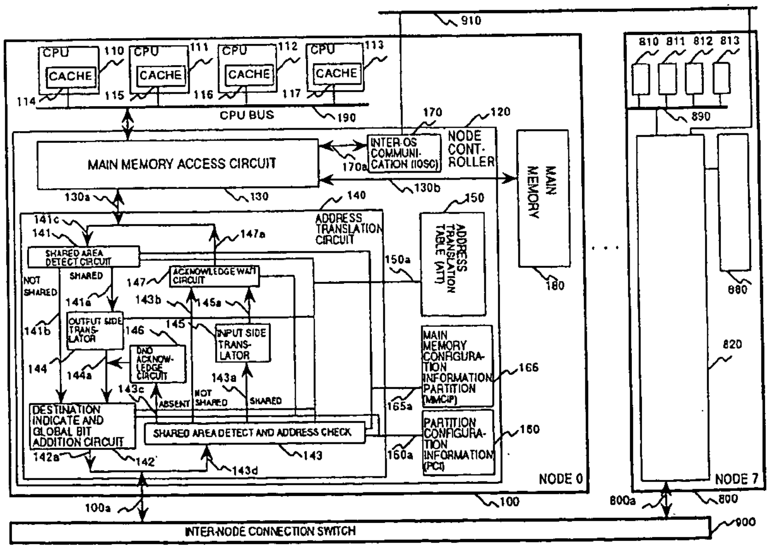


FIG. 1

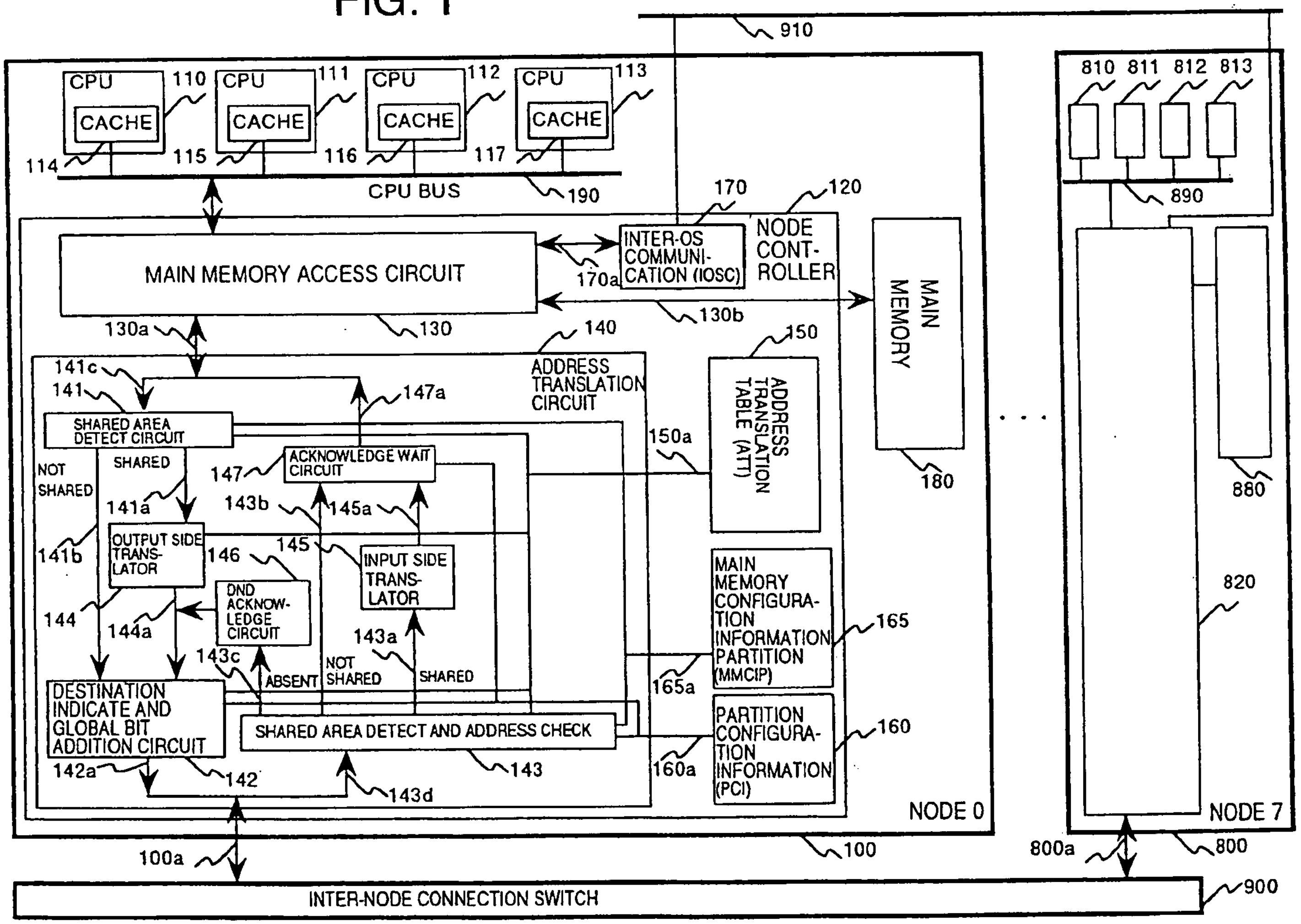

910
CPU
CACHE
110
CPU
CACHE
111
CPU
CACHE
112
CPU
CACHE
113
114
115
116
117
CPU BUS
190
170
120
NODE CONT-ROLLER
MAIN MEMORY ACCESS CIRCUIT
INTER-OS COMMUNI-CATION (IOSC)
170a
MAIN MEMORY
130a
130
130b
140
ADDRESS TRANSLATION CIRCUIT
150
ADDRESS TRANSLATION TABLE (ATT)
180
141c
141
147a
SHARED AREA DETECT CIRCUIT
150a
NOT SHARED
SHARED
147
ACKNOWLEDGE WAIT CIRCUIT
141a
143b
145a
OUTPUT SIDE TRANS-LATOR
146
145
INPUT SIDE TRANS-LATOR
141b
DND ACKNOW-LEDGE CIRCUIT
MAIN MEMORY CONFIGURA-TION INFORMATION PARTITION (MMCIP)
165
144
144a
143a
143c
NOT SHARED
ABSENT
SHARED
165a
DESTINATION INDICATE AND GLOBAL BIT ADDITION CIRCUIT
SHARED AREA DETECT AND ADDRESS CHECK
PARTITION CONFIGURA-TION INFORMATION (PCI)
160
160a
142a
142
143
143d
NODE 0
100a
100
INTER-NODE CONNECTION SWITCH
900
810
811
812
813
890
880
820
NODE 7
800a
800

FIG. 2

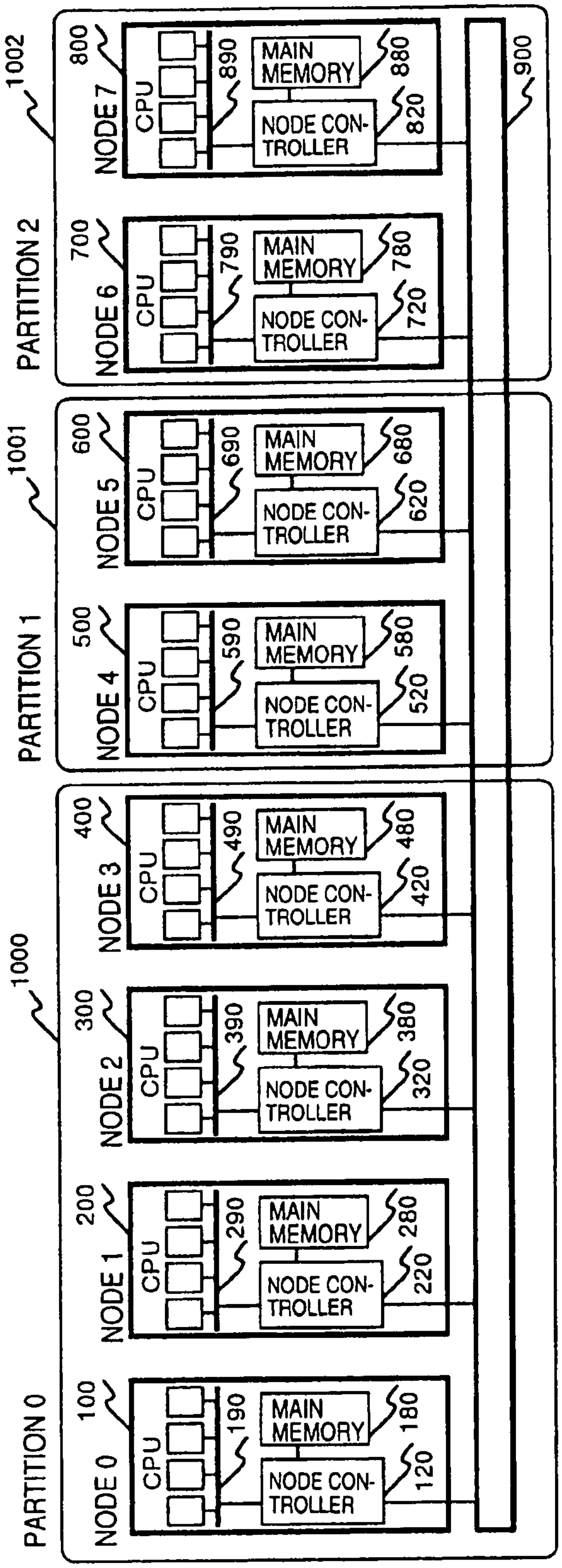

PARTITION 0
PARTITION 1
PARTITION 2
1000
1001
1002
NODE 0
NODE 1
NODE 2
NODE 3
NODE 4
NODE 5
NODE 6
NODE 7
100
200
300
400
500
600
700
800
CPU
MAIN MEMORY
NODE CON-TROLLER
190
180
120
290
280
220
390
380
320
490
480
420
590
580
520
690
680
620
790
780
720
890
880
820
900

FIG. 3

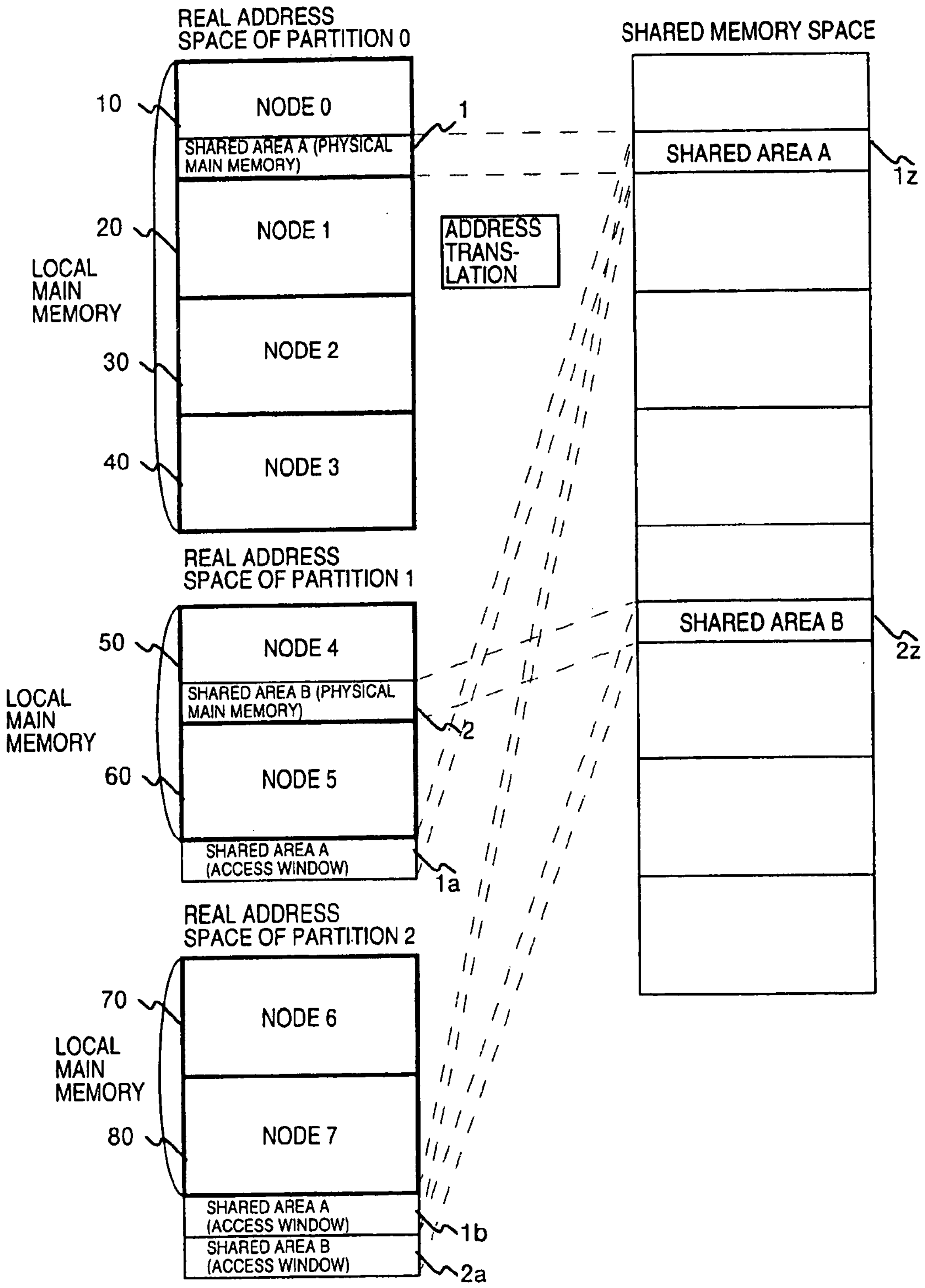
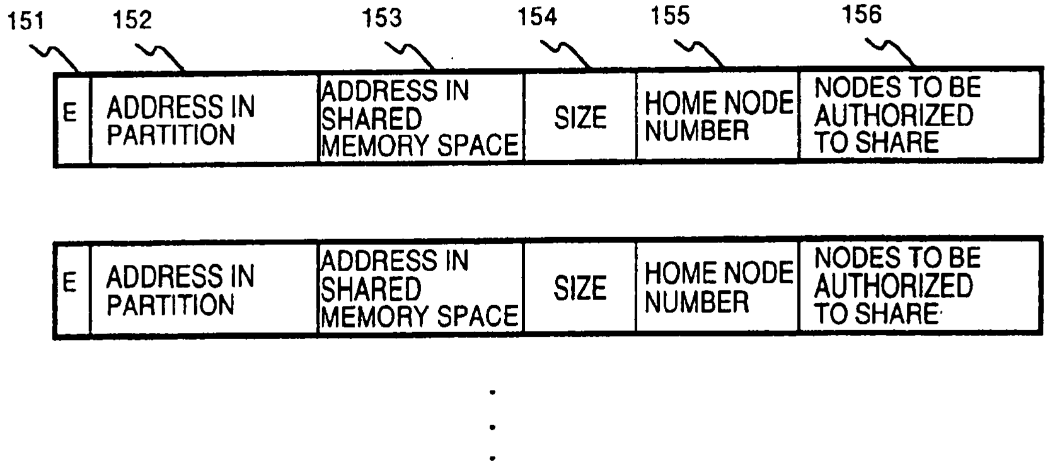

REAL ADDRESS SPACE OF PARTITION 0
SHARED MEMORY SPACE
10
NODE 0
1
SHARED AREA A (PHYSICAL MAIN MEMORY)
SHARED AREA A
1z
20
NODE 1
ADDRESS TRANS-LATION
LOCAL MAIN MEMORY
NODE 2
30
40
NODE 3
REAL ADDRESS SPACE OF PARTITION 1
50
NODE 4
SHARED AREA B
2z
SHARED AREA B (PHYSICAL MAIN MEMORY)
2
LOCAL MAIN MEMORY
60
NODE 5
SHARED AREA A (ACCESS WINDOW)
1a
REAL ADDRESS SPACE OF PARTITION 2
70
NODE 6
LOCAL MAIN MEMORY
80
NODE 7
SHARED AREA A (ACCESS WINDOW)
1b
SHARED AREA B (ACCESS WINDOW)
2a

FIG. 4

#	SITUATION		ADDRESS IN INTERNAL BUS	ADDRESS IN THE SWITCH	TRANSLATION METHOD FROM/ INTO ADDRESS IN THE SWITCH	DESTINATION OF SNOOP COMMAND	ERROR DETECTION
1	NOT SHARED		ADDRESS IN MAIN MEMORY OF PARTITION	ADDRESS IN PARTITION	NO TRANSLATION	NODES IN OWN PARTITION	ACCESS SOURCE IS IN OTHER PARTITION
2	SHARED	HOME NODE BELONGS TO OWN PARTITION (EXPORTING SIDE)	ADDRESS IN MAIN MEMORY OF PARTITION	ADDRESS IN SHARED MEMORY SPACE	OPEN AN AREA WHICH IS A SHARED IN MAIN MEMORY OF OWN PARTITION, ON SHARED MEMORY SPACE	ALL NODES	ACCESS SOURCE IS NOT AUTHORIZED TO SHARE
3		HOME NODE DOES NOT BELONG TO OWN PARTITION (IMPORTING SIDE)	ADDRESS OF ACCESS WINDOW FOR ACCESSING MAIN MEMORY OF PARTITIONS	ADDRESS IN SHARED MEMORY SPACE	MAP AN AREA ON SHARED MEMORY SPACE, THE AREA HAVING BEEN OPENED AS #2, TO AN AREA FOR WHICH MAIN MEMORY OF OWN PARTITION IS NOT SUPPLIED	ALL NODES	ACCESS SOURCE IS NOT AUTHORIZED TO SHARE

FIG. 5



E	ADDRESS IN PARTITION	ADDRESS IN SHARED MEMORY SPACE	SIZE	HOME NODE NUMBER	NODES TO BE AUTHORIZED TO SHARE
E	ADDRESS IN PARTITION	ADDRESS IN SHARED MEMORY SPACE	SIZE	HOME NODE NUMBER	NODES TO BE AUTHORIZED TO SHARE

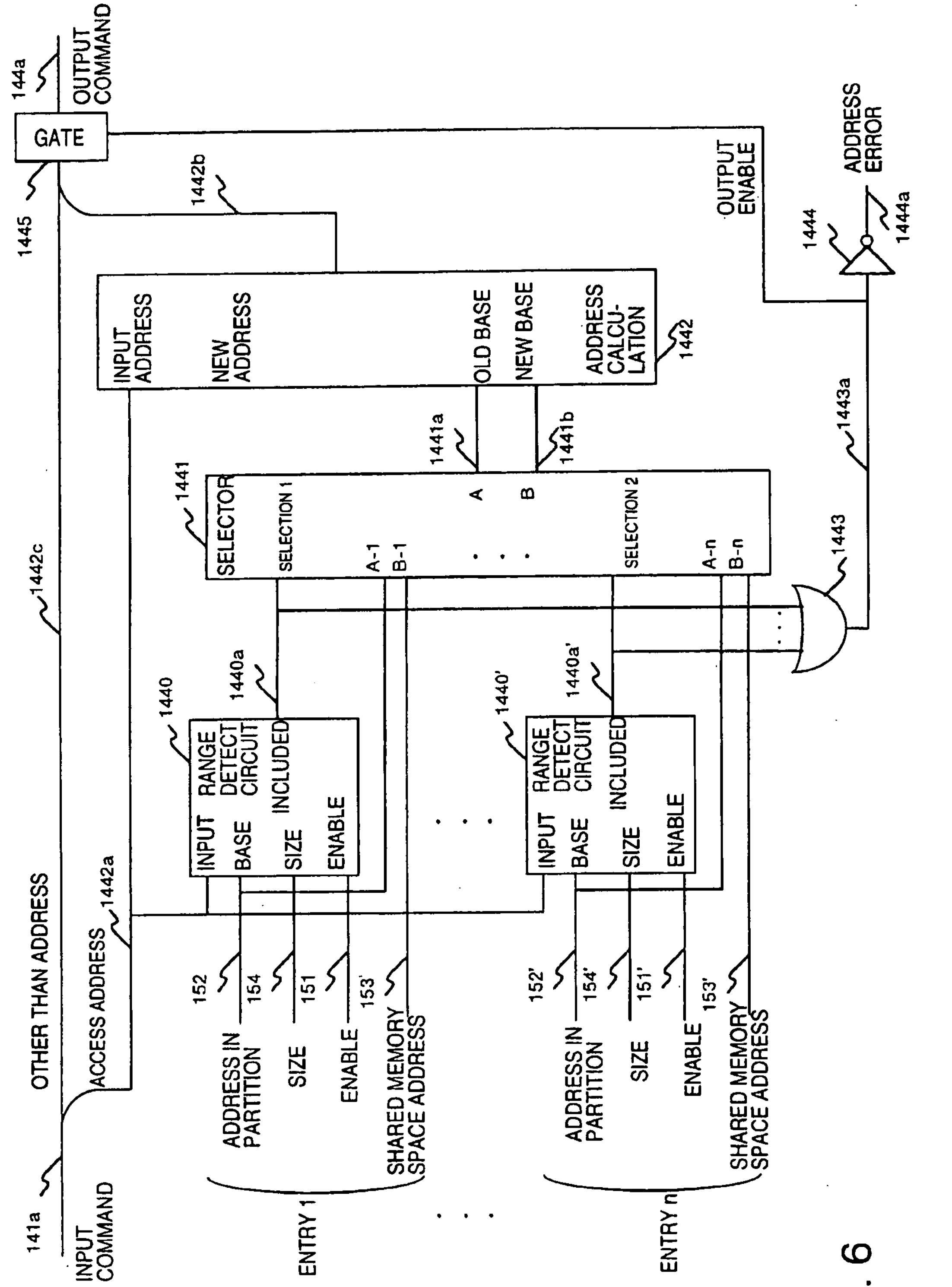
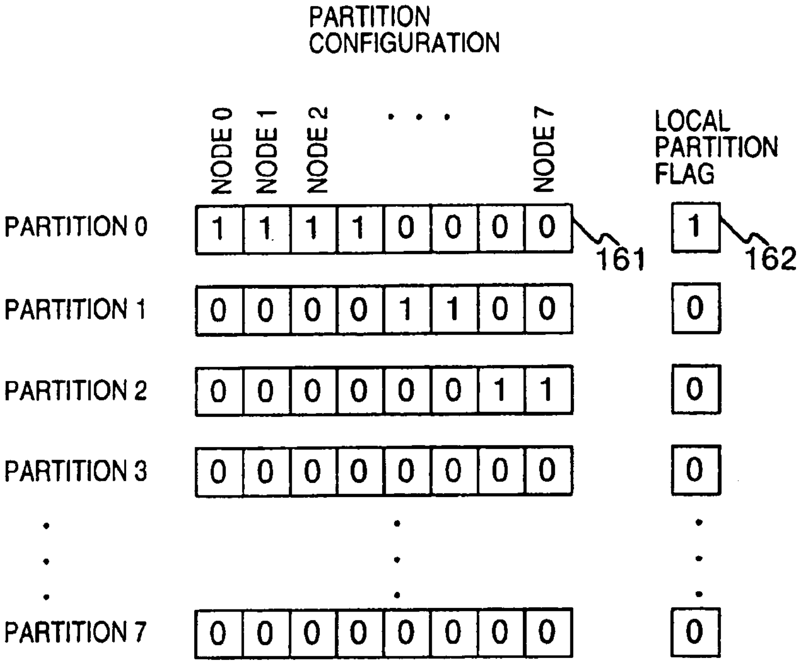

OUTPUT COMMAND
144a
GATE
1445
1442b
OUTPUT ENABLE
ADDRESS ERROR
1444
1444a
INPUT ADDRESS
NEW ADDRESS
OLD BASE
NEW BASE
ADDRESS CALCU-LATION
1442
1441a
1441b
1443a
1441
SELECTOR
SELECTION 1
A
B
A-1
B-1
SELECTION 2
A-n
B-n
1443
1442c
1440
1440a
RANGE DETECT CIRCUIT
INPUT
BASE
INCLUDED
SIZE
ENABLE
1440'
1440a'
RANGE DETECT CIRCUIT
INPUT
BASE
INCLUDED
SIZE
ENABLE
OTHER THAN ADDRESS
ACCESS ADDRESS
1442a
152
154
151
153'
ADDRESS IN PARTITION
SIZE
ENABLE
SHARED MEMORY SPACE ADDRESS
152'
154'
151'
153'
ADDRESS IN PARTITION
SIZE
ENABLE
SHARED MEMORY SPACE ADDRESS
141a
INPUT COMMAND
ENTRY 1
ENTRY n

FIG. 6

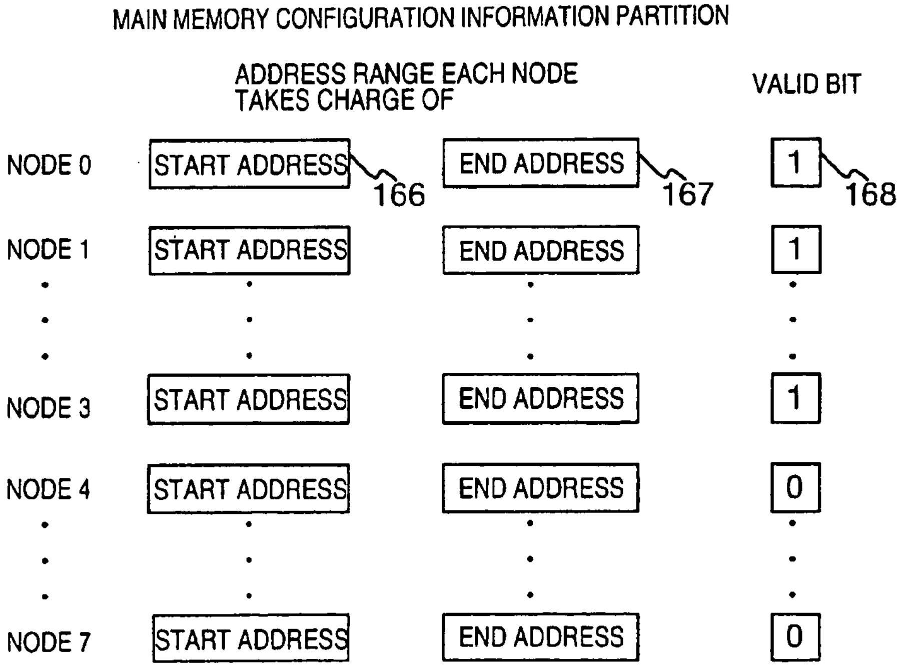
FIG. 7



PARTITION CONFIGURATION

	NODE 0	NODE 1	NODE 2	· · ·				NODE 7	LOCAL PARTITION FLAG
PARTITION 0	1	1	1	1	0	0	0	0	1
PARTITION 1	0	0	0	0	1	1	0	0	0
PARTITION 2	0	0	0	0	0	0	1	1	0
PARTITION 3	0	0	0	0	0	0	0	0	0
⋮				⋮					⋮
PARTITION 7	0	0	0	0	0	0	0	0	0

FIG. 8



MAIN MEMORY CONFIGURATION INFORMATION PARTITION

	ADDRESS RANGE EACH NODE TAKES CHARGE OF		VALID BIT
NODE 0	START ADDRESS	END ADDRESS	1
NODE 1	START ADDRESS	END ADDRESS	1
⋮	⋮	⋮	⋮
NODE 3	START ADDRESS	END ADDRESS	1
NODE 4	START ADDRESS	END ADDRESS	0
⋮	⋮	⋮	⋮
NODE 7	START ADDRESS	END ADDRESS	0

FIG. 9

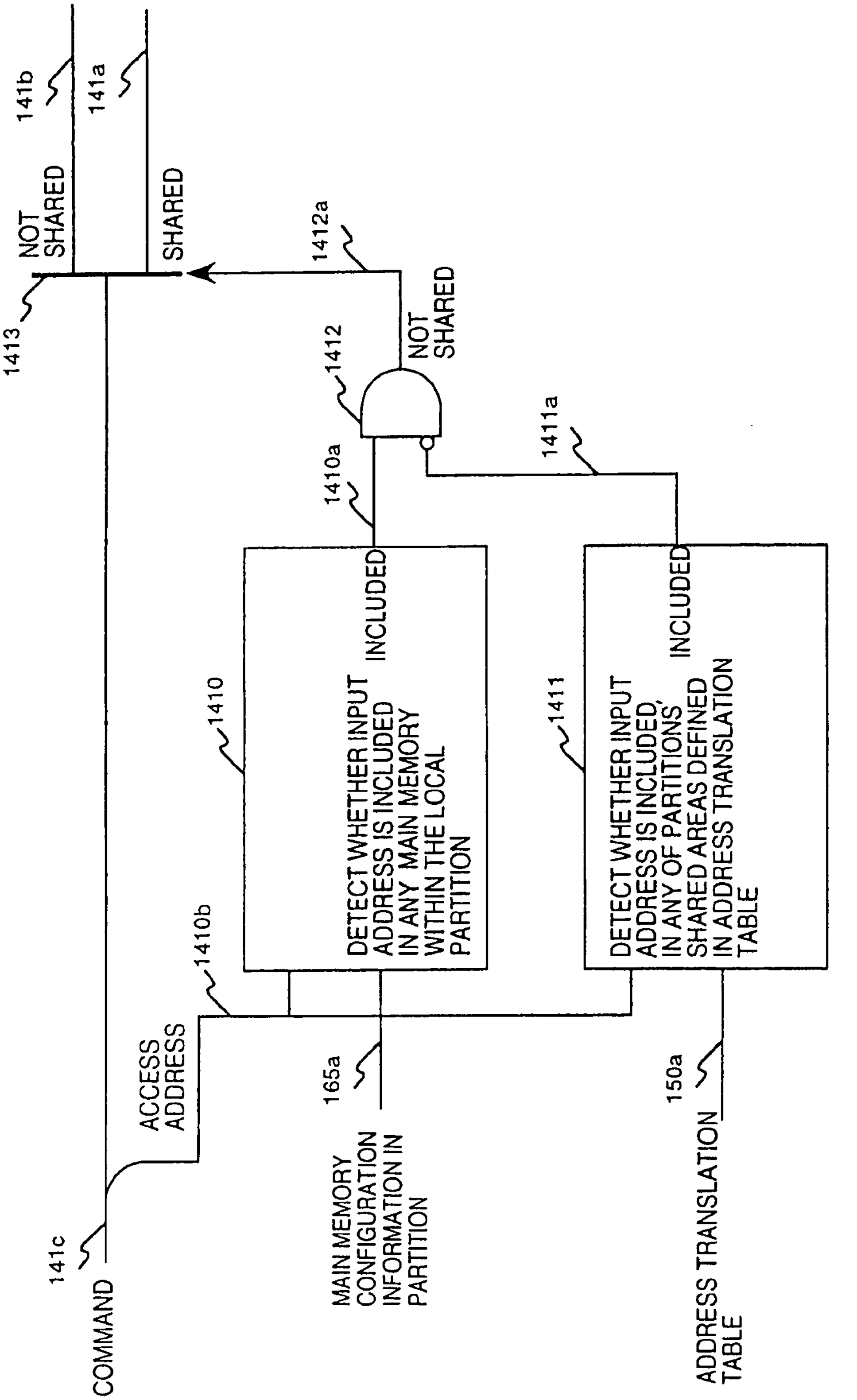

141b
141a
NOT SHARED
SHARED
1413
1412a
NOT SHARED
1412
1410a
1411a
INCLUDED
INCLUDED
1410
DETECT WHETHER INPUT ADDRESS IS INCLUDED IN ANY MAIN MEMORY WITHIN THE LOCAL PARTITION
1411
DETECT WHETHER INPUT ADDRESS IS INCLUDED IN ANY OF PARTITIONS' SHARED AREAS DEFINED IN ADDRESS TRANSLATION TABLE
1410b
ACCESS ADDRESS
165a
150a
141c
COMMAND
MAIN MEMORY CONFIGURATION INFORMATION IN PARTITION
ADDRESS TRANSLATION TABLE

FIG. 10

#	SITUATION		COMMAND ON NOT-SHARED ADDRESS (INPUT FROM 141b)	COMMAND ON SHARED ADDRESS (INPUT FROM 144b)
1	REQUEST	SNOOP COMMAND (F,FI,I)	MULTICAST IN PARTITION GLOBAL BIT = 0	BROADCAST IN ALL NODES GLOBAL BIT = 1
2		MAIN MEMORY ACCESS (WB,UR,UW)	SEND TO HOME NODE IN LOCAL PARTITION, INDICATED IN PARTITION CONFIGURATION INFORMATION GLOBAL BIT= 0	SEND TO HOME NODE, INDICATED IN ADDRESS TRANSLATION TABLE GLOBAL BIT= 1
3	ACKNOWLEDGE (D,DM,ND,DND)		RETURN TO REQUESTING NODE GLOBAL BIT = 0	RETURN TO REQUESTING NODE GLOBAL BIT = 1

FIG. 11

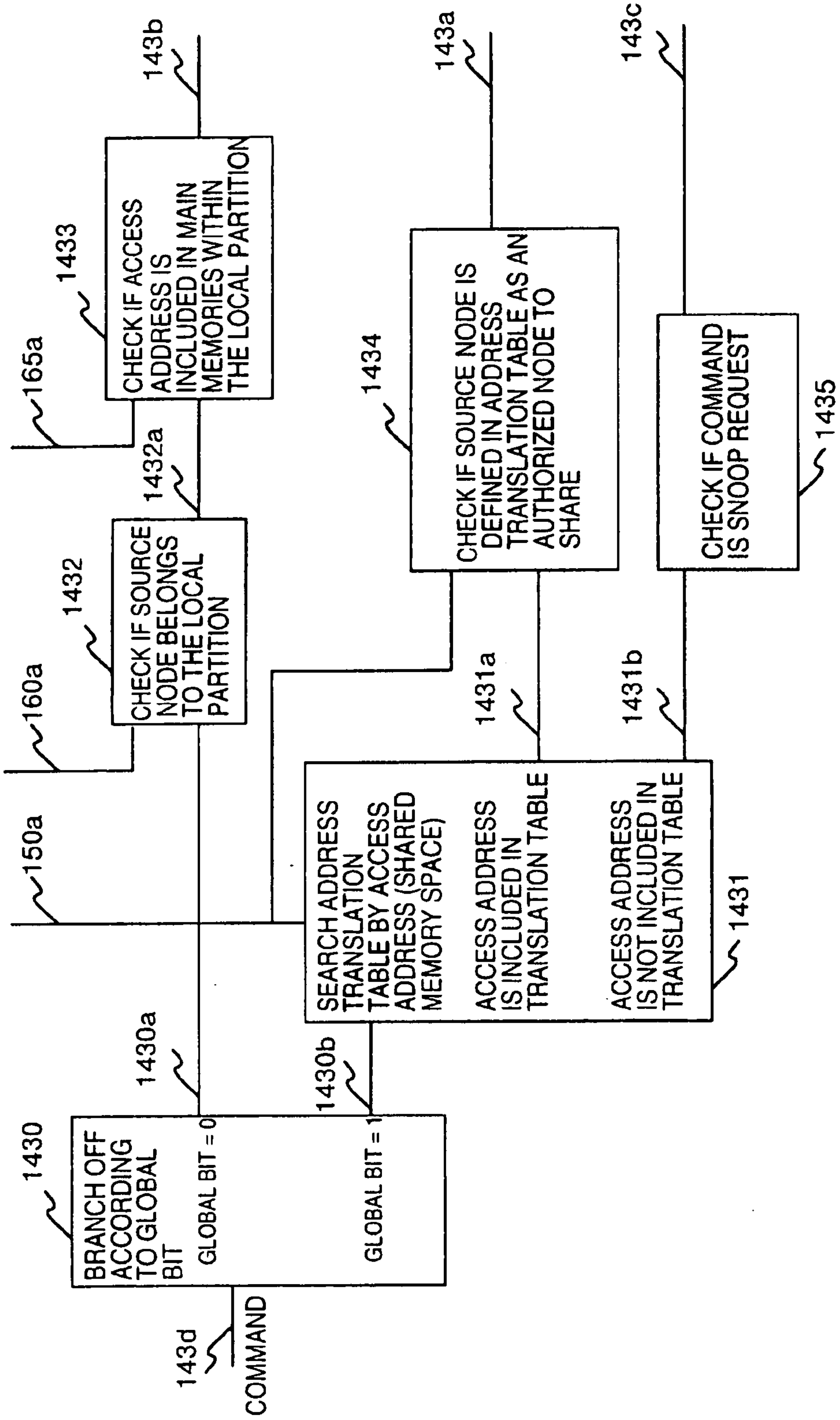

COMMAND
143d
BRANCH OFF ACCORDING TO GLOBAL BIT
1430
GLOBAL BIT = 0
GLOBAL BIT = 1
1430a
1430b
150a
160a
165a
CHECK IF SOURCE NODE BELONGS TO THE LOCAL PARTITION
1432
1432a
CHECK IF ACCESS ADDRESS IS INCLUDED IN MAIN MEMORIES WITHIN THE LOCAL PARTITION
1433
143b
SEARCH ADDRESS TRANSLATION TABLE BY ACCESS ADDRESS (SHARED MEMORY SPACE)
ACCESS ADDRESS IS INCLUDED IN TRANSLATION TABLE
ACCESS ADDRESS IS NOT INCLUDED IN TRANSLATION TABLE
1431
1431a
1431b
CHECK IF SOURCE NODE IS DEFINED IN ADDRESS TRANSLATION TABLE AS AN AUTHORIZED NODE TO SHARE
1434
143a
CHECK IF COMMAND IS SNOOP REQUEST
1435
143c

FIG. 12

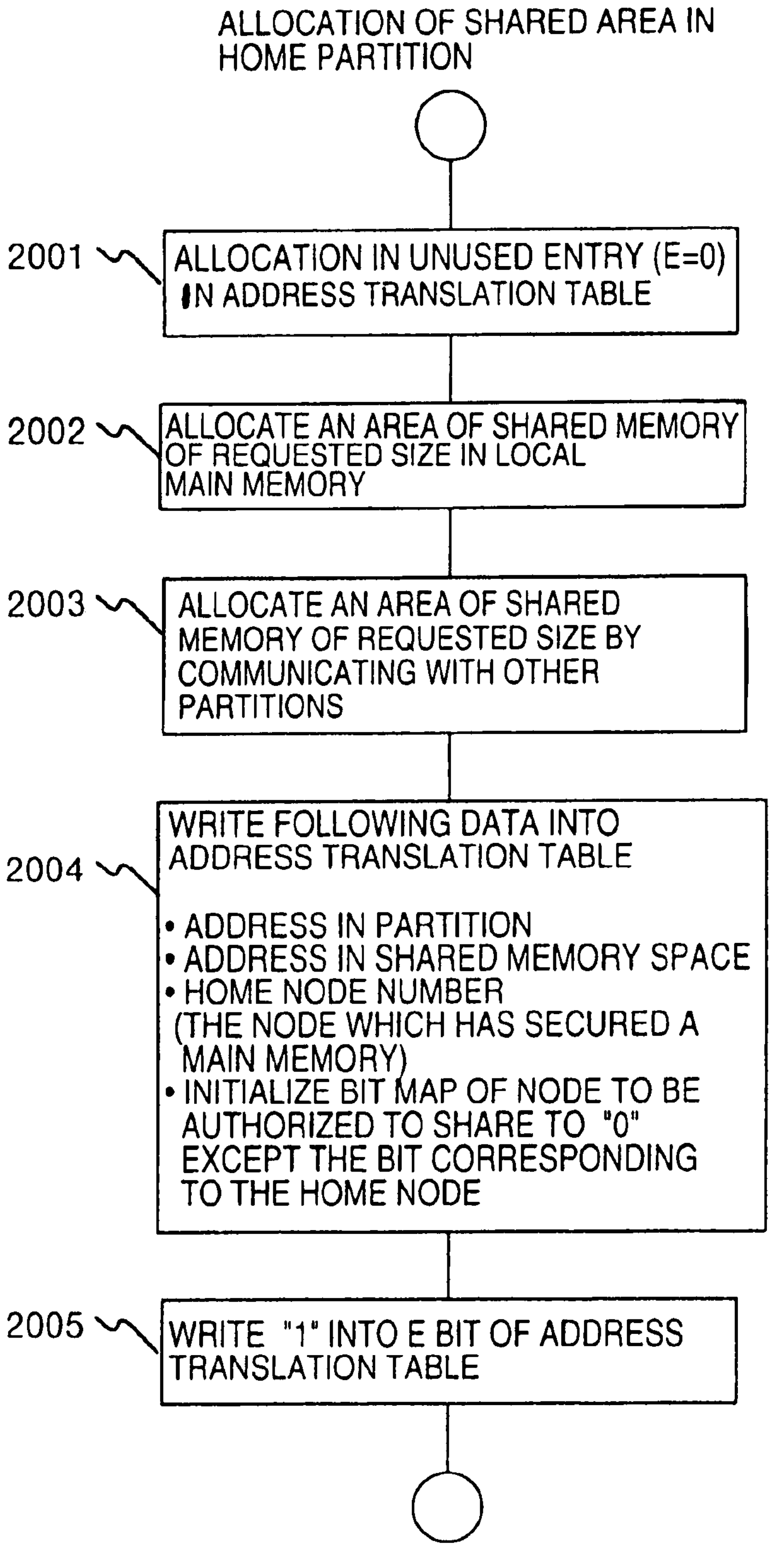

ALLOCATION OF SHARED AREA IN HOME PARTITION
2001
ALLOCATION IN UNUSED ENTRY (E=0) IN ADDRESS TRANSLATION TABLE
2002
ALLOCATE AN AREA OF SHARED MEMORY OF REQUESTED SIZE IN LOCAL MAIN MEMORY
2003
ALLOCATE AN AREA OF SHARED MEMORY OF REQUESTED SIZE BY COMMUNICATING WITH OTHER PARTITIONS
2004
WRITE FOLLOWING DATA INTO ADDRESS TRANSLATION TABLE
• ADDRESS IN PARTITION
• ADDRESS IN SHARED MEMORY SPACE
• HOME NODE NUMBER (THE NODE WHICH HAS SECURED A MAIN MEMORY)
• INITIALIZE BIT MAP OF NODE TO BE AUTHORIZED TO SHARE TO "0" EXCEPT THE BIT CORRESPONDING TO THE HOME NODE
2005
WRITE "1" INTO E BIT OF ADDRESS TRANSLATION TABLE

FIG. 13

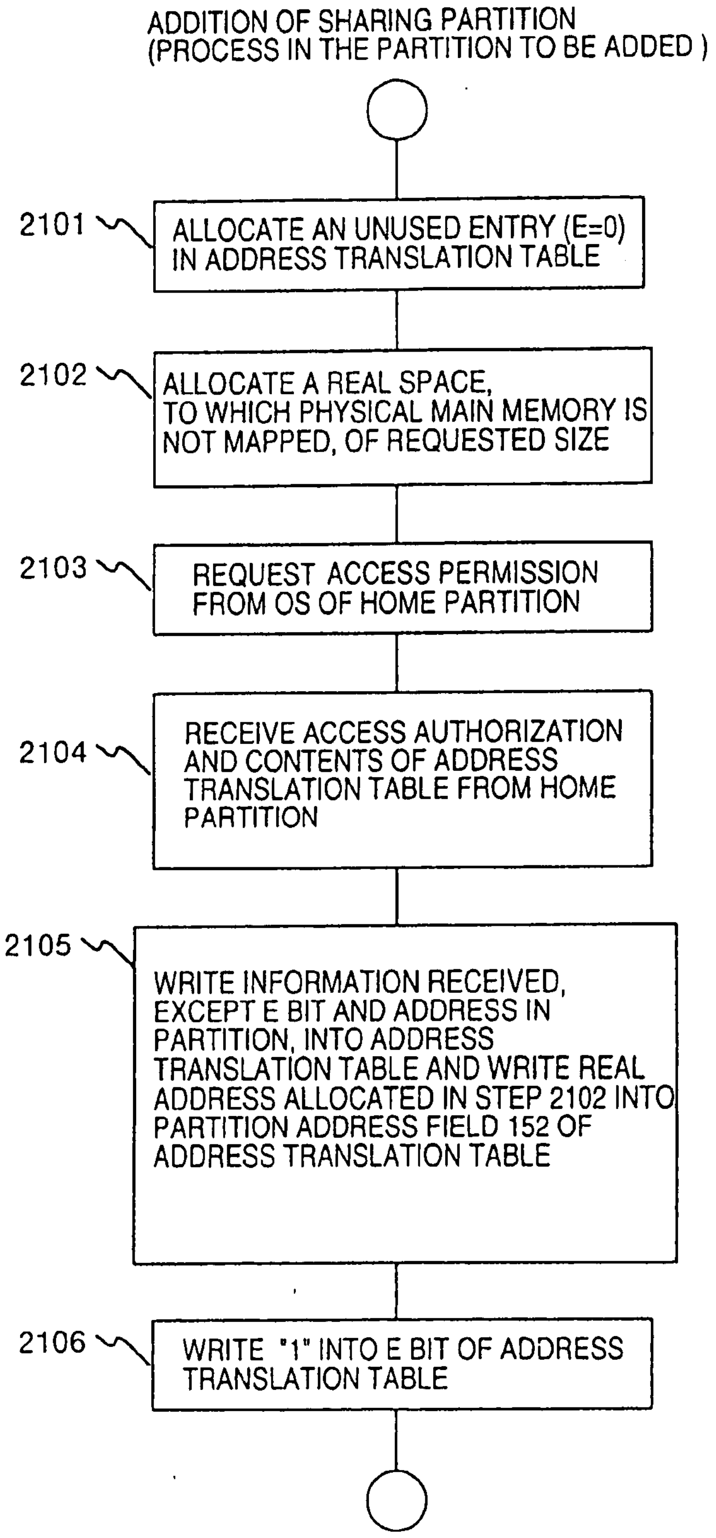

ADDITION OF SHARING PARTITION
(PROCESS IN THE PARTITION TO BE ADDED)
2101
ALLOCATE AN UNUSED ENTRY (E=0) IN ADDRESS TRANSLATION TABLE
2102
ALLOCATE A REAL SPACE, TO WHICH PHYSICAL MAIN MEMORY IS NOT MAPPED, OF REQUESTED SIZE
2103
REQUEST ACCESS PERMISSION FROM OS OF HOME PARTITION
2104
RECEIVE ACCESS AUTHORIZATION AND CONTENTS OF ADDRESS TRANSLATION TABLE FROM HOME PARTITION
2105
WRITE INFORMATION RECEIVED, EXCEPT E BIT AND ADDRESS IN PARTITION, INTO ADDRESS TRANSLATION TABLE AND WRITE REAL ADDRESS ALLOCATED IN STEP 2102 INTO PARTITION ADDRESS FIELD 152 OF ADDRESS TRANSLATION TABLE
2106
WRITE "1" INTO E BIT OF ADDRESS TRANSLATION TABLE

FIG. 14

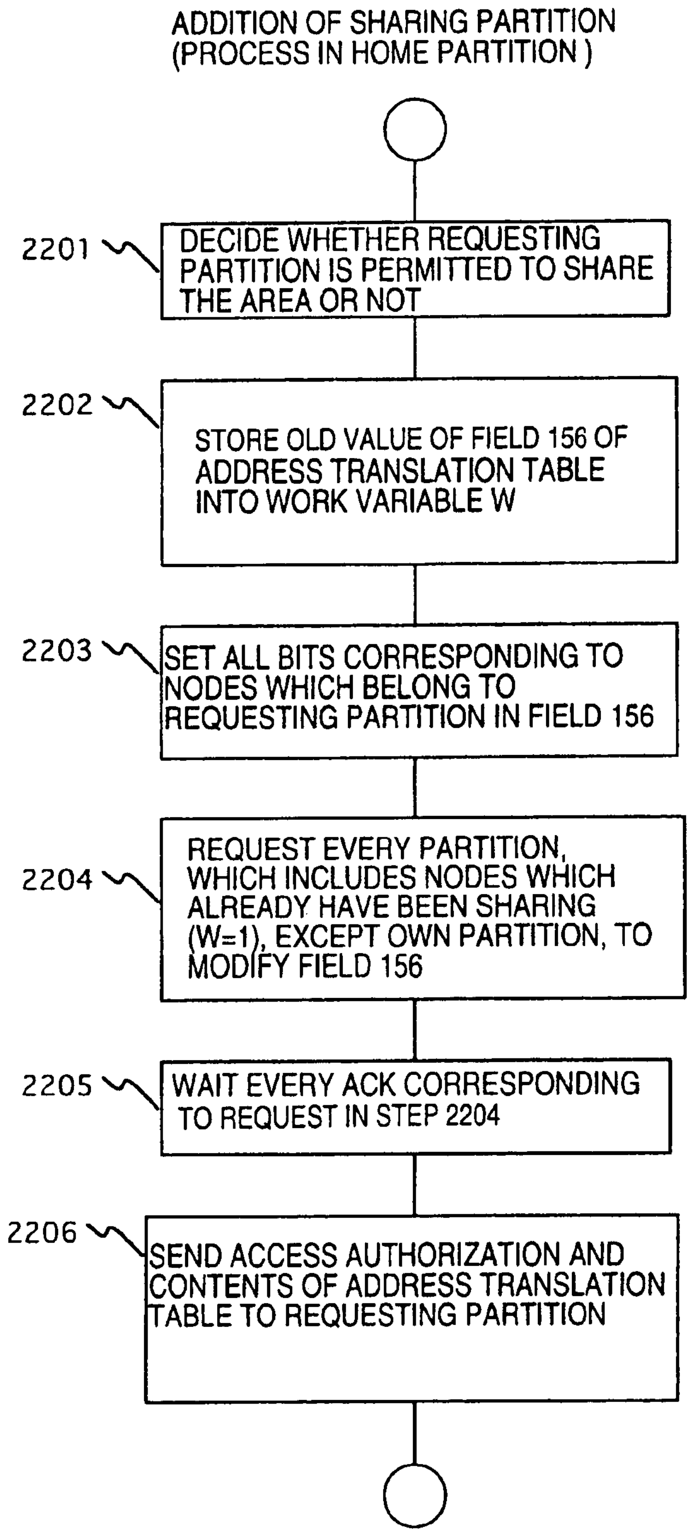

ADDITION OF SHARING PARTITION
(PROCESS IN HOME PARTITION)
2201
DECIDE WHETHER REQUESTING PARTITION IS PERMITTED TO SHARE THE AREA OR NOT
2202
STORE OLD VALUE OF FIELD 156 OF ADDRESS TRANSLATION TABLE INTO WORK VARIABLE W
2203
SET ALL BITS CORRESPONDING TO NODES WHICH BELONG TO REQUESTING PARTITION IN FIELD 156
2204
REQUEST EVERY PARTITION, WHICH INCLUDES NODES WHICH ALREADY HAVE BEEN SHARING (W=1), EXCEPT OWN PARTITION, TO MODIFY FIELD 156
2205
WAIT EVERY ACK CORRESPONDING TO REQUEST IN STEP 2204
2206
SEND ACCESS AUTHORIZATION AND CONTENTS OF ADDRESS TRANSLATION TABLE TO REQUESTING PARTITION

FIG. 15

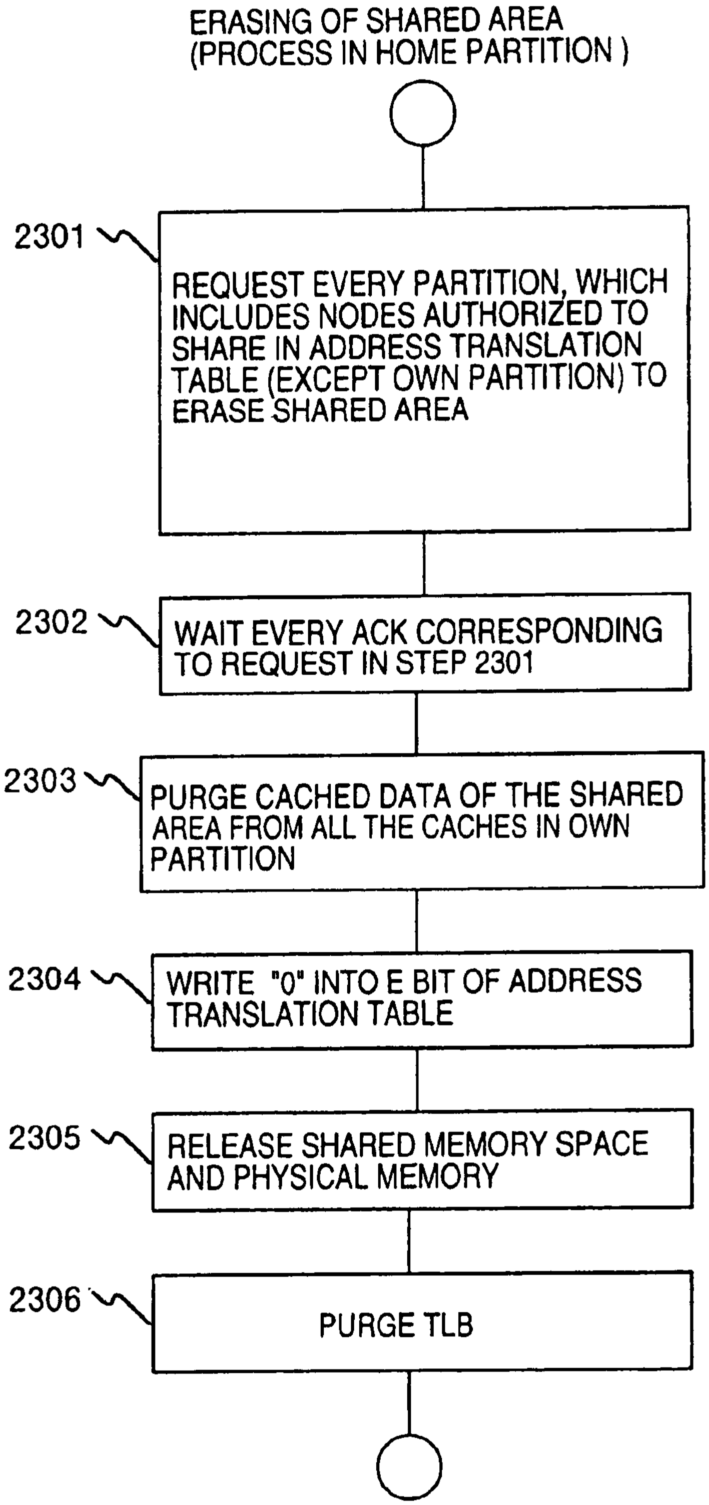

ERASING OF SHARED AREA
(PROCESS IN HOME PARTITION)
2301
REQUEST EVERY PARTITION, WHICH INCLUDES NODES AUTHORIZED TO SHARE IN ADDRESS TRANSLATION TABLE (EXCEPT OWN PARTITION) TO ERASE SHARED AREA
2302
WAIT EVERY ACK CORRESPONDING TO REQUEST IN STEP 2301
2303
PURGE CACHED DATA OF THE SHARED AREA FROM ALL THE CACHES IN OWN PARTITION
2304
WRITE "0" INTO E BIT OF ADDRESS TRANSLATION TABLE
2305
RELEASE SHARED MEMORY SPACE AND PHYSICAL MEMORY
2306
PURGE TLB

FIG. 16

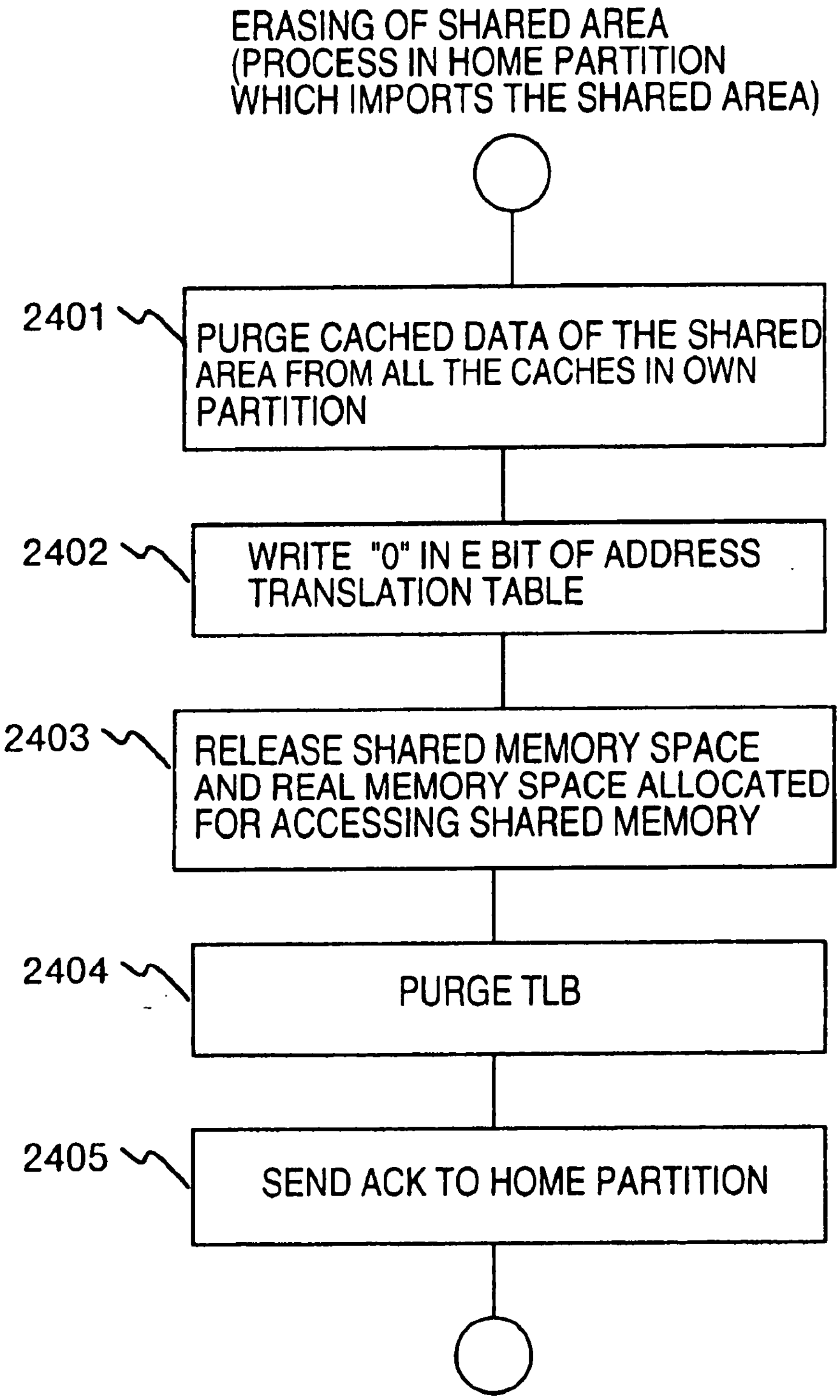
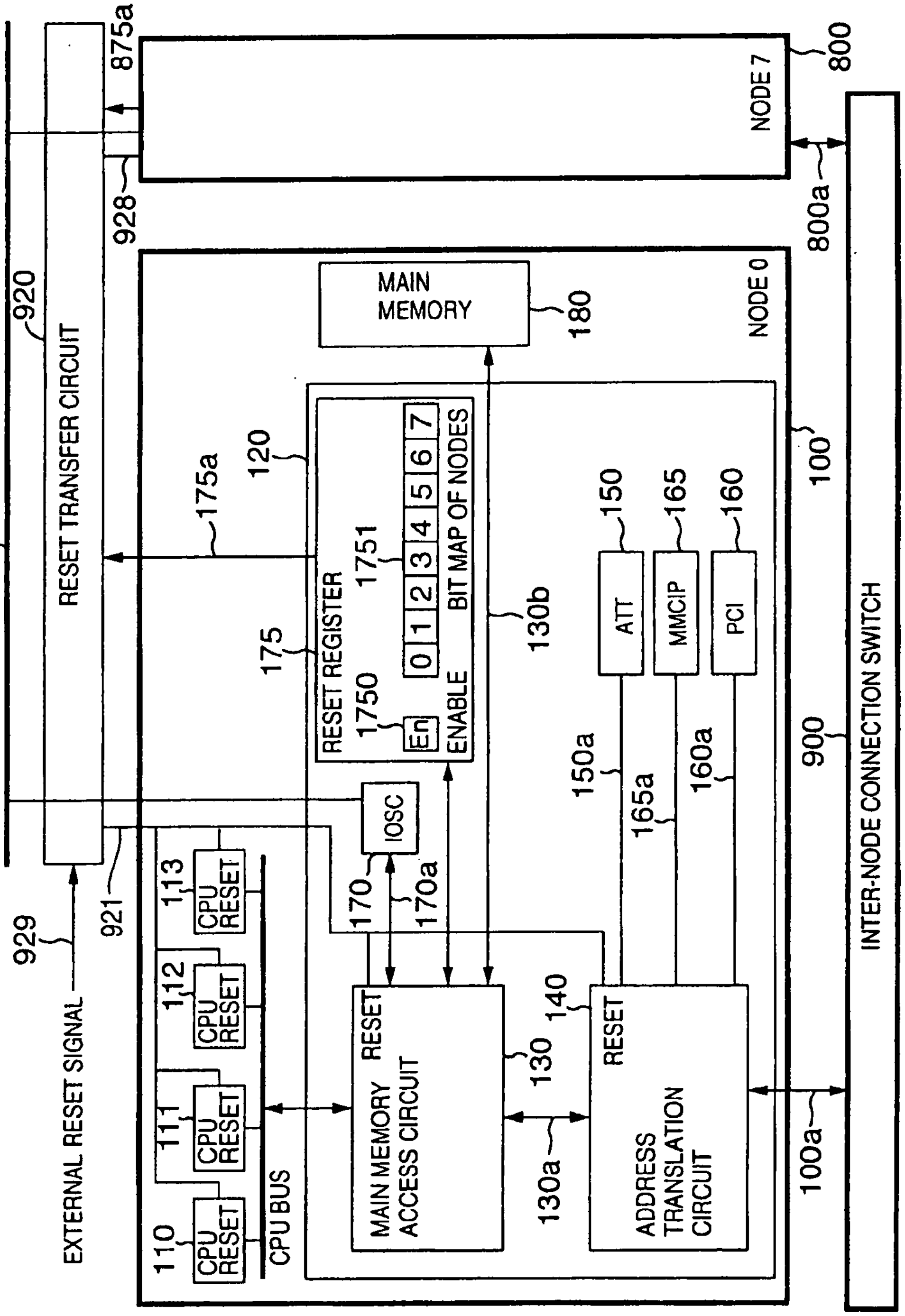

ERASING OF SHARED AREA
(PROCESS IN HOME PARTITION
WHICH IMPORTS THE SHARED AREA)
2401
PURGE CACHED DATA OF THE SHARED AREA FROM ALL THE CACHES IN OWN PARTITION
2402
WRITE "0" IN E BIT OF ADDRESS TRANSLATION TABLE
2403
RELEASE SHARED MEMORY SPACE AND REAL MEMORY SPACE ALLOCATED FOR ACCESSING SHARED MEMORY
2404
PURGE TLB
2405
SEND ACK TO HOME PARTITION

FIG. 17


EXTERNAL RESET SIGNAL
929
921
910
RESET TRANSFER CIRCUIT
920
928
875a
NODE 7
800
800a
NODE 0
MAIN MEMORY
180
100
120
175a
RESET REGISTER
175
1751
0 1 2 3 4 5 6 7
BIT MAP OF NODES
1750
En
ENABLE
130b
ATT
150
MMCIP
165
PCI
160
150a
165a
160a
IOSC
170
170a
CPU RESET
110
CPU RESET
111
CPU RESET
112
CPU RESET
113
CPU BUS
MAIN MEMORY ACCESS CIRCUIT
RESET
130
130a
140
RESET
ADDRESS TRANSLATION CIRCUIT
100a
900
INTER-NODE CONNECTION SWITCH

SHARED MEMORY MULTIPROCESSOR SYSTEM AND METHOD WITH ADDRESS TRANSLATION BETWEEN PARTITIONS AND RESETTING OF NODES INCLUDED IN OTHER PARTITIONS

CROSS-REFERENCE TO RELATED APPLICATION

The present application relates to subject matter described in application Ser. No. 09/030,957 filed Feb. 26, 1998 entitled "Shared Memory Multiprocessor", now U.S. Pat. No. 6,088,770, the disclosure of which is hereby incorporated by reference.

BACKGROUND OF THE INVENTION

The present invention relates to a shared memory parallel processor system used in information processing apparatuses, especially used in personal computers (PCs), work stations (WSs), and server machines. In particular, the present invention relates to a control scheme of a shared memory between partitions.

In recent years, use of the architecture of the shared memory multiprocessor as a host module of the parallel processors has spread. In this architecture, such a configuration where several tens to several hundreds processors share a main memory is needed in some cases in order to improve the performance. As the configuration method of the shared memory multiprocessor, bus connection symmetrical multiprocessors (SMPs) used in personal computers are typical. Since the bus throughput forms a bottleneck in the bus connection SMPs, however, the number of processors which can be connected is limited to approximately four. Thus the bus connection SMP is not suitable for such a scheme as to connect a large number of processors.

In order to solve the above described problem, there has been proposed a method of connection bus connecting SMPs hierarchically by using a crossbar switch or the like. A typical example of the hierarchical SMP is found in "Gigaplane—XB: Extending the Ultra Enterprise Family", HOT Interconnects V, pp. 97 to 112, August 1997. The crossbar switch or the like between nodes logically functions as a bus. Coherence of a CPU cache between nodes of the bus connection SMP having processors and main memories can be managed at high speed by using a bus snoop protocol.

As one of problems of the large scale shared memory multiprocessor as described above, there is reliability. In conventional shared memory multiprocessors, the whole system has one Operation System (OS). Since all processors of the system can be managed by one OS, this scheme has an advantage that flexible system operation (such as load distribution) can be conducted. However, this scheme has a drawback that the system reliability falls in the case where a large number of processors are connected in a shared memory multiprocessor configuration. In a server of a cluster configuration in which a plurality of processors are connected by a network, respective nodes have different OSs. Even if a fatal error such as a bug or the like of an OS or the like occurs, only the corresponding node suffers from a system down state. On the other hand, if a certain processor is brought into a down state by a system bug or the like in the case where the whole system is controlled by one OS in a shared memory multiprocessor, the OS is brought into a down state and consequently all processors are affected.

In order to solve this problem, there has been proposed such a scheme that the inside of a shared memory multiprocessor is divided into a plurality of partitions and a plurality of OSs are run independently. Each partition has an independent main memory. A processor of a certain partition basically accesses only the main memory of its own partition. As a result, it becomes possible to realize the fault containment between partitions and improve the system performance.

Furthermore, also for improving the operation performance and reducing the management cost using server consolidation, it is desired to integrate works which have been conducted by a plurality of servers into one highly multiplexed server. The above described partition technique is indispensable.

In the case where a shared memory multiprocessor is divided into partitions, how communication is conducted between partitions poses a problem. A scheme in which communication between partitions is conducted by making efficient use of a shared memory mechanism provided in a system before partitioning is at advantage in performance. Therefore, realization of a shared memory between partitions becomes necessary.

A partition technique of making a plurality of OSs run in one system has been used heretofore in mainframes, and it has been disclosed in U.S. Pat. No. 4,843,541. In this scheme, it is possible to make a plurality of guest OSs operate under the management of a host OS which manages the whole system. Respective guest OSs are independent systems having different address spaces. Access to a main memory in each partition is conducted according to the following procedure.

(1) A virtual address of a guest is translated to a real address.

(2) The above described guest real address is translated to a main memory address in the host.

(3) The main memory is accessed by using the main memory address in the host derived in (2).

The above described address translation of the two stages must be conducted between a CPU and the main memory.

In the partitions of the main frame, it is made possible for respective guest partitions to have different address spaces and the fault containment is realized by conducting the above described address translation of the two stages. By overlapping addresses of guests in the address translation of (3), the shared memory can be realized.

In realizing a partition mechanism and an inter-partition shared memory of a hierarchical bus connection SMP by using the above described conventional techniques, there are problems described hereafter.

The conventional inter-partition shared memory mechanism is premised on a concentrated main memory architecture having an address translation mechanism of two stages between each CPU and the main memory. Therefore, the conventional inter-partition shared memory mechanism is largely different in architecture from the hierarchical bus connection SMP. Accordingly, the conventional technique cannot be applied to the hierarchical bus connection SMP as it is. In particular, respective CPUS use standard components. As a result, the address translation of the two stages used in the conventional technique cannot be conducted in the CPU, and relocation of the address of each partition (guest) cannot be conducted.

Furthermore, in the hierarchical SMP, the CPU cache coherence is kept at a high speed by using the bus snoop protocol. Therefore, the inter-partition shared memory mechanism needs to be capable of supporting the bus snoop protocol.

SUMMARY OF THE INVENTION

Therefore, an object of the present invention is to realize a partition mechanism and an inter-partition shared memory mechanism suitable for the architecture of the hierarchical SMP.

In addition, a future parallel system must support a general purpose OS. Accordingly, the partition system needs to have a general purpose architecture which does not depend on a specific OS. It is necessary to make it possible for each partition to have a free address space. In addition, it is necessary to realize dynamic generation and erasing of a partition in order to deal with a large number of applications and improve the reliability of the system by using dynamic reconfiguration of partitions.

Another object of the present invention is to flexibly manage the configuration of the inter-partition shared memory.

In addition, the partition system needs to realize high reliability at a low cost. Thus it is indispensable for partitions to back up each other. Therefore, a third object of the present invention is to facilitate recovery from an error from another partition in the case where the OS of a certain partition suffers from system down.

In order to achieve the above described first and second objects, such a hierarchical SMP that nodes each having CPUs coupled by a bus and a main memory are connected by a switch and cache coherence control is conducted through the switch, at the gateway of the switch from each node when the inside of the system is divided into partitions in each of which a different OS operates, with means for mutually translating an address of an access command for an area shared between partitions, between a real address used in a partition and an address used in common between partitions. As a result, the address of a local area of each partition is freely set. In addition, cache coherence control of the shared area can be conducted at high speed by using a snoop command of the hierarchical SMP.

Furthermore, in another preferred aspect of the present invention, conformity between the address of the access command issued from another partition and the configuration of the shared area is checked at the gateway of each node. As a result, fault containment can be realized between partitions.

Furthermore, in another preferred aspect of the present invention, there is provided apparatus for the system software to dynamically modify the configuration information of the shared area between partitions. As a result, flexible management of the shared area becomes possible.

In addition, in order to achieve the above described third object, each partition is provided with a function of resetting CPUs of other partitions. In the case where a certain partition suffers from system down, it is possible to reset and re-initialize the partitions which have suffered from system down.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a shared memory multiprocessor having a partition mechanism according to the present invention;

FIG. 2 shows an example of partitioning in a shared memory multiprocessor according to the present invention;

FIG. 3 shows an example of address spaces of respective partitions in a shared memory multiprocessor according to the present invention;

FIG. 4 shows a summary table of relations between access addresses and operations of address translation circuits of respective nodes;

FIG. 5 shows the configuration of an address translation table of each node;

FIG. 6 shows a detailed diagram of an output side translator of each node;

FIG. 7 shows the configuration of partition configuration information of respective nodes;

FIG. 8 shows the configuration of main memory configuration information in partition of respective nodes;

FIG. 9 shows a detailed diagram of a shared area detect circuit of each node;

FIG 10 shows a table describing relations between access commands and a destination indicate and global bit addition circuit of respective nodes;

FIG. 11 shows a detailed diagram of a shared area detect and address check circuit of each node;

FIG. 12 shows a flow diagram showing a process conducted when a home partition ensures a sa in memory in a multiprocessor system of the present invention;

FIG. 13 shows a flow diagram showing a process conducted on an added partition side when a shared partition is added in a multiprocessor system of the present invention;

FIG. 14 shows a flow diagram showing a process conducted on a home partition side when a shared partition is added in a multiprocessor system of the present invention;

FIG. 15 shows a flow diagram showing a process conducted on a home partition side when a shared area is erased in a multiprocessor system of the present invention;

FIG. 16 shows a flow diagram showing a process conducted on a side of a partition which imports the shared area when a shared area is erased in a multiprocessor system of the present invention; and

FIG. 17 shows the details of a reset circuit in a multiprocessor system of the present invention.

DESCRIPTION OF THE EMBODIMENTS

FIG. 1 is a block diagram of a shared memory multiprocessor according to the present invention. The present system is an 8-node system. A plurality of nodes **100** to **800** (which are sometimes referred to as nodes **0** to **7**) are connected by a bus or internode connection switch **900**. Six nodes other than the nodes **100** and **800** are omitted in FIG. **1** in order to avoid the complexity. However, every node has the same structure. By taking the node **0** as a representative example, the structure will be described. The node has a main memory **180** and four processors, i.e., Central Processor Units (CPUs) **110, 111, 112** and **113**. These CPUs are coupled to a node controller **120** via a CPU bus **190**. The node controller **120** includes a main memory access circuit **130**, an address translation circuit **140**, partition configuration information **160**, and main memory configuration information partition **165**. The main memory **180** forms a part of a main memory common to this system, and holds a part of a program executed by each node and data. This system is the so-called parallel processor system of distributed shared memory type.

The cache coherence between CPUs in the node is managed by a bus snoop protocol. As the bus snoop technique, a known technique is employed. In FIG. **1**, the CPUS in the node are connected by the bus. As for the hardware, however, a coupling scheme other than the bus, such as one-to-one coupling or coupling using a switch, may also be used. As for these connection methods in the nodes, various known techniques can be applied.

The cache coherence between the CPUs of the nodes is also managed by using the bus snoop protocol. The internode connection switch **900** logically functions in the same way as the bus.

The address translation circuit **140** is hardware for translation between an address in the node (an address used by the CPUs **110** to **113**, the CPU bus **190**, and the main memory **180**) and an address used by the inter-node connection switch **900** (an address outside the node).

Within the address translation circuit **140**, the side for outputting a command from the inside of the node includes a shared area detect circuit **141** for determining whether the accessed area is a shared area, an output side translator **144** for translating the address of the shared area, and a destination indicate and global bit addition circuit **142** for sending out a command to proper destination. The side for inputting a command from the outside of the node includes a shared area detect and address check circuit **143** for determining whether a command from another node is a command directed to a shared area and checking the address, an input side translator **145** for translating the address of a shared area, a DND acknowledge circuit **146** for issuing an acknowledge in response to an access request for a shared area which is not shared by its own node, and an acknowledge wait circuit **147** for waiting an acknowledge of a snoop command from another node. These circuits performs operations specific to the present invention.

Each node has an inter-OS communication function **170** to be used for system initialization process, configuration control, debug and so on. Without using the shared memory, system software of each node can perform communication. Physically, an inter-OS communication path **910** can also share the same hardware as the inter-node connection switch. Alternatively, the inter-OS communication path **910** can also use a typical network such as a LAN.

FIG. 2 shows a configuration example of partitioning in a hierarchical SMP of the present invention. Partitioning is conducted by taking a SMP node as the unit, and an arbitrary combination of nodes can be adopted. In the example shown in FIG. 2, the inside of the system is divided into three partitions. A first partition **1000** includes nodes **0** to **3** (**100**, **200**, **300** and **400**). A second partition **1001** includes nodes **4** and **5** (**500** and **600**). A third partition **1002** includes nodes **6** and **7** (**700** and **800**). Respective partitions are independent systems in that different OSs operate. Respective partitions have independent address spaces as hereafter described. In addition, partitions have a shared memory which is a part of the main memory and which is shared by partitions.

In the case where there is not a memory shared by partitions, exchange of data between partitions must be conducted by using a message communication mechanism or a typical network (LAN). Since the latency of message communication requires approximately at least 10 μs, it is difficult to realize high speed communication. If a hierarchical bus connection shared memory mechanism is utilized, however, communication between nodes can be realized in several hundreds ns or less. Therefore, the inter-partition shared memory is indispensable for realizing faster communication between partitions, faster resource lock processing between partitions, and faster fail over between partitions.

FIG. 3 shows an example of address maps of respective partitions in the case where a shared memory mechanism of partitions is used. This example corresponds to the partitioning exemplified in FIG. 2.

Respective partitions (partitions **0** to **2**) have real address spaces which are independent from partition to partition (as shown on the left hand side of FIG. 3). Respective nodes in the same partition have the same real address space. The real address of the above described partition is the address used in the CPU bus **190** in nodes included in the partition. It is a real address to be used by the CPUs **110** to **113**, the main memory **180** and the main memory access circuit **130** which

are included in the node. In addition to the above described real address spaces of respective partitions, the system has a shared memory space (shown on the right hand side of FIG. **3**) common to the whole system. An area shared by partitions have different addresses within the partitions, but has a common address in the shared memory space.

Access from each partition to a shared area is conveyed to another partition via a shared memory space address. In other words, an access command such as a cache coherent check command (CCC command) issued to a shared area by each node is transformed from the real address of its own partition to a shared memory space address when the access command is outputted from the accessing node. When the access command is inputted to an accessed node, the access command is transformed from the shared memory space address to a real address of an accessed partition. These address transformation operations are conducted by the address translation circuit **140** located at the gateway of each node. Owing to this address translation function, it is possible in each partition to freely determine a real address space which exceeds the range of the physical main memory of a node in its own partition.

It should be noted that in the case where the bus snoop protocol is used there is a possibility that a CCC command will be issued by an arbitrary CPU which shares a certain cache line and the CCC command might be broadcasted in some cases to all CPUS which may possibly share the pertinent line. Therefore, address translation conducted in each partition between the local real address of a partition and the shared space address must be translatable in both directions. Furthermore, also when accessing a shared area which has a physical main memory in the same partition (such as when one of the nodes **0** to **3** accesses a shared area A), an address in the CCC command must be translated to an address of the shared memory space once in order to correctly convey the CCC command to nodes located outside the partition as well. For access to a not-shared area (an area which is accessed by only nodes included in the partition), address translation is not conducted at all (the CCC command is sent to only the nodes included in the partition). At that time, it is impossible to determine on the basis of the address alone whether a command inputted from the switch is access to a not-shared space or access to a space shared by partitions. Therefore, each command in the switch has a global bit whereby it can be determined whether the command is access to a shared space.

In accordance with a feature of the present invention, the address translation circuit **140** is provided in the gateway of each node to the switch, and mutual translation between the address of a shared area in each node and its shared memory space address is conducted.

Hereafter, an address management scheme between partitions will be described in detail by referring to the example shown in FIG. 3. In FIG. **3**, the address space of the first partition **1000** has local main memories **10**, **20**, **30** and **40** respectively of the nodes **0** to **3**. The address space of the second partition **1001** has local main memories **50** and **60** respectively of the nodes **4** and **5** and a shared area **1***a*. The address space of the third partition **1002** has local main memories **70** and **80** respectively of the nodes **6** and **7** and shared areas **1***b* and **2***a*. All of the nodes included in each partition have the same address space. For example, both the nodes **4** and **5** can access the areas **50**, **60** and **1***a*.

As for shared areas, there are two shared areas A and B. The shared area A is allocated onto the local main memory of the node **0**, and the shared area B is allocated onto the local main memory of the node **4**. They are represented as shared area A (physical main memory) **1** and shared area B (physical main memory) **2**, respectively. Partitions having main memories corresponding to these shared areas are hereafter referred to as partitions of export side. On the other hand, in the shared memory space, the shared area A and the shared area B are located in **1***z* and **2***z*, respectively. Therefore, the partition **0** maps (exports) the shared area A from the real address space **1** in the partition to the shared memory space **1***z*, and the partition **1** maps (exports) the shared area B from the real address space **2** in the partition to the shared memory space **2***z*.

Furthermore, the shared area A is shared by the partition **1**, the partition **2** and the partition **3**. The shared area B is shared by the partition **2** and the partition **3**. Like the partitions **2** and **3** for the shared area A and the partition **3** for the shared area B, a partition which has not a main memory of the pertinent area therein, i.e., a partition which accesses a physical main memory of the shared area included in another partition, is referred to as partition of import side. In the partition of the import side, an area (window) for accessing the main memory area included in another partition is formed in a portion of its own partition where a main memory of a real address is not mounted. Therefore, the partition 1 maps (imports) the shared area A from the shared memory space **1***z* to the real address space **1***a* in the partition. The partition **2** maps (imports) the shared area A from the shared memory space **1***z* to the real address space **1***b* in the partition, and maps (imports) the shared area B from the shared memory space **2***z* to the real address space **2***a* in the partition.

The foregoing description is summarized as follows. In the example of FIG. 3, the following (bidirectional) address translation operations are required at gateways of respective nodes.

The shared area A is exported from the main memory **1** in the node **0** to the shared memory space **1***z*.

Nodes **4** and **5** (the Second Partition **1001**)

The shared area B is exported from the main memory **2** in the node **4** to the shared memory space **2***z*.

The shared area A is imported from the shared memory space **1***z* to the area **1***a* in its own partition.

Nodes **6** and **7** (the third partition **1002**)

The shared area A is imported from the shared memory space **1***z* to the area **1***b* in its own partition.

The shared area B is imported from the shared memory space **2***z* to the area **2***a* in its own partition.

The address of each of not-shared areas which are not mentioned in the foregoing description is not translated even when a CCC command is outputted from a node.

FIG. 4 collectively shows address translation functions required at respective gateways of respective nodes in the present invention. Real addresses are divided into two classes, i.e., not-shared address (area which can be accessed from only the inside of a partition) and shared address (address shared by partitions). In addition, the shared addresses are divided into two classes, i.e., the case where there is a main memory in its own partition, and the case where there is not a main memory in its own partition.

(1) Not-shared Area

As for the not-shared area which is accessed from only the inside of the partition, the address in partition is used in the switch as well. Therefore, address translation at the gateway of the node is not conducted. The snoop command is multicast only to nodes in the partition.

(2) Shared Area (Export Side)

As for a shared area having a physical main memory in the partition, translation between the address of the main memory in the own partition and the address of the shared memory space in the switch is required. The snoop command is sent to all nodes. Such optimization that the snoop command is sent to only nodes of sharing partitions is also possible.

(3) Shared Area (Import Side)

As for a shared area having no physical main memory in the partition, translation between the address of a window for accessing a shared area in the own partition and the address of a shared memory space in the switch is required. The snoop command is sent to all nodes. In this case as well, such optimization that the snoop command is sent to only nodes of sharing partitions is also possible.

Furthermore, in order to realize the fault containment between partitions, unauthorized access between partitions is checked. In ordinary highly multiplexed SMP, access authority check is conducted at the time of address translation when the CPU translates a virtual address to a real address. Assuming that the system software of an OS or the like of another partition runs away, however, access authority check on the CPU side which issues the access is not enough, but it is necessary to conduct access authority check in the accessed partition. Furthermore, since the cache coherence management using the bus snoop protocol is conducted between CPUs, not only data in the main memory but also data cached in the cache must be protected from unauthorized access. In order to realize the above described check, the following check is conducted at the entrance of each node on a CCC command which has arrived from another node.

(1) Not-shared Area

Commands from nodes located outside the partition are not authorized.

(2) Shared Area (Both the Export Side and the Import Side)

In the address translation hardware, there is provided means for storing nodes authorized to share each shared area (i.e., nodes sharing the area) in a bit map form. Commands from nodes which are not authorized to share the shared area are not accepted.

The address translation heretofore described is dynamically set by system software such as an OS or middle software when the shared memory is allocated. The scheme of setting will be described later.

Hereafter, the configuration of a shared memory multiprocessor which realizes the inter-partition shared memory mechanism and the inter-partition address translation heretofore described will be described in detail.

With reference to FIG. **1**, the address translation circuit **140** is disposed at the gateway of each node, to be concrete, between the main memory access circuit **130** and the inter-node connection switch **900**. The address translation circuit **140** is a circuit for conducting address translation inside and outside the node, destination specification of a CCC command sent to the outside of the node, and error check for a CCC command sent from the outside of the node, on the basis of information written in the address translation table **150**, the partition configuration information **160**, and the main memory configuration information in partition **165**.

The address translation table **150**, the partition configuration information **160**, and the main memory configuration information in partition **165** have been subjected to memory mapping. The processors in the partitions can access them.

FIG. **5** shows the format of an entry of the address translation table **150**. In order to conduct address translation on the shared area, the address translation table has one entry shown in FIG. **5** per shared area. The address translation circuit conducts full-associative bidirectional address translation. Therefore, the number of shared areas exported and imported by a certain node is limited by the number of entries of the address translation table. As a result, the number of entries of the address translation table must be sufficiently large. Each entry has the following information.

"Enable bit" (E) **151**

The entry is made valid.

"Address in partition" **152**

"Address in shared memory space" **153**

(These two addresses are respective start addresses.)

"Size" **154**

Size of shared space

"Home node number" **155**

Number of a node having the pertinent shared area in the main memory

"Nodes to be authorized to share" **156**

A list of node numbers authorized to share the pertinent area (stored in a bit map form).

(Whether nodes are authorized to share is determined by taking a partition as the unit. With due regard to the processing efficiency, storage is conducted in the present table by taking a node number as the unit.)

Here, the shared memory is handled every certain fixed unit (such as 1 MB). In that case, fields **152** to **154** become multiples of 1 MB. Therefore, twenty low-order bits of the fields **152** to **154** are fixed to **0**. Bits of the table are not mounted.

FIG. **7** shows details of the partition configuration information **160**. The partition configuration information **160** shows how the system is partitioned. The partition configuration information **160** is formed of a bit map **161** showing which nodes are included in which partition, and a flag **162** showing which partition its own node is housed. The number of entries of the table is the total number of nodes of the system. Therefore, up to the case of one node/one partition can be supported. FIG. **7** shows values of the partition configuration information **160** in the nodes **0** to **3** illustrated in FIG. **2**. In other words, in this example, the partition **0** is its own partition, and the partitions **3** to **7** are not used. In other nodes as well, the same information is stored except the local partition flag **162**.

FIG. **8** shows details of the main memory configuration information in partition **165**. The main memory configuration information in partition indicates the address range the local main memory of each of the nodes included in the own partition takes charge of. The main memory configuration information in partition does not have information of other partitions. The number of entries is the total number of nodes of the system. Up to the case where the whole system is formed of one partition can be supported. The main memory configuration information in partition includes a start address **166** and an end address **167** of the main memory each node takes charge of, and a V bit **168** indicating that the entry is valid.

Commands exchanged by the inter-node connection switch **900** will now be described. In the hierarchical SMP, data reading/writing (cache coherent check: CCC) of each CPU is managed by the bus snoop protocol. As for the bus protocol in the node, a known technique is adopted. As for the CCC between nodes as well, a known technique is used. Here, as an example of the CCC between nodes, outline of a procedure of the bus snoop using a fetch command between nodes will now be described. For the purpose of description, it is now assumed that partitioning is not conducted. In other words, it is now assumed that the whole system is formed of one partition.

(1) In the case where a certain CPU reads data, the fetch command is issued to all other nodes (all CPUs and the main memory of the home node).

(2) If there is modified data (latest data) in the cache in the own node, the node which has received the fetch command returns that data. Otherwise, the node which has received the fetch command replies that there are no modified data. If there are no modified data in the cache, the home node returns data in the main memory.

(3) In the accessing node, replies from other nodes are totalized. If modified data in the cache is sent from another node, the data is returned to the accessing CPU. If every node does not have modified data, data returned from the main memory (data of the main memory in the own node in the case where the own node is the home node) is returned to the accessing CPU. For determining whether data returned from the main memory should be used, it is necessary to wait the reply "there are no data" from all nodes.

Hereafter, CCC commands used on the bus within the nodes of the hierarchical SMP will be described. Characters enclosed in parentheses are abbreviations used in the embodiment.

Fetch (F)

It requests line transfer of data.

It is issued in the case where a read command of the CPU has failed.

Fetch&Invalidate (FI)

It requests invalidation of data on other caches simultaneously with the line transfer of data.

It is issued in the case where a read command of the CPU has failed.

Invalidate (I)

It requests invalidation of data on other caches. It is issued in the case where the CPU has issued a write request to a cache line shared with other caches.

WriteBack (WB)

It requests writeback of a cache line.

It is issued when data has been turned out by replacement.

Data (D)

It is a command for returning data in response to the F or FI command in the case where there is modified (latest) data in the cache.

DataMem (DM)

It is a command for returning data in the main memory of the home node in response to the F or FI command. In the home node, it is returned in the case where there are no modified (latest) data in the cache within the node. In the case where a D command has come from any other node, data returned by the DM command is ignored. (Data in the cache is given priority.)

It is also used for the reply to UncachedRead.

NoData (ND)

It is returned in response to the F or FI command in the case where there are no modified (latest) data in the cache of the pertinent node (except the home node).

DummyNoDatä (DND)

A reply returned from a node located outside the partition in response to the F or FI command.

(DND is not an ordinary CCC command, but it is a command required in implementation specific to the present embodiment. Its meaning is the same as that of ND.)

UncachedRead (UR)
UncachedWrite (UW)

It is a command for directly accessing the main memory in cache off.

A command in the inter-node connection switch has the following fields specific to the present invention besides fields required for CCC, such as command, address, and data.

(1) Destination Node Number

The destination node number is represented by a bit map. By virtue of representation of destination using a bit map, it is possible to easily realize multicast directed to a plurality of specific nodes, such as nodes in the partition, by setting a plurality of bits and further broadcast directed to all processors of the system by setting all bits.

(2) Global Bit

It is a bit for determining whether the pertinent access command is a command for a shared area (hence the address is an address of the shared memory space) or a command for a not-shared area (hence the address is a local address in the partition).

Hereafter, operations of the address translation circuit in the case where a command is issued to another node, and in the case where a command has been received from another node will be described separately for respective cases and successively.

(A) Processing in the Case where a Command is Issued from a Node

If a CCC command is delivered from the main memory access circuit **130** to the address translation circuit, it is first inputted to the shared area detect circuit **141**.

FIG. **9** shows details of the shared area detect circuit **141**. From an inputted command **141***c*, a part of an access address **1410***b* is taken out. The access address **1410***b* is inputted to circuits **1410** and **1411**.

In the circuit **1410**, it is detected by using the main memory configuration information in partition **165***a* whether the access address **1410***b* is included in any of main memories within the local partition or not. In other words, it is checked for every bit having the valid bit **168** of FIG. **8** set equal to 1 whether the access address **1410***b* is located between the start address **166** and the end address **167**. If the access address is included in any of the main memories, then "1" is outputted to output **1410***a* and the access address **1410***b* is judged to be an address having a main memory in the partition. Since the internal circuit is the same as that of FIG. **6**, detailed description thereof will be omitted.

In the circuit **1411**, it is detected by using the information **150***a* of the address translation table whether the access address **1410***b* is included in any of partitions' shared areas defined in the address translation table or not. In other words, it is checked for every bit having the E bit of FIG. **5** set equal to "1" whether the access address **1410***b* is included between the "address in partition" **152** and the "size" **154**. If the access address is included in any of the partitions' shared areas, then "1" is outputted to output **1411***a*, and the access address **1410***b* is judged to be one of shared areas exported from the inside of the partition or imported from the inside of the partition.

If a gate **1412** outputs "1", i.e., if the access address **1410***b* has a main memory in the partition, but it is not a shared area exported or imported, then the access command is distributed to a not-shared side **141***b* by a gate **1413**. Otherwise (i.e., if the output of the gate **1412** is "0"), i.e., the access address **1410***b* does not have a main memory in the partition, or it is an exported or imported shared area, then the access command is distributed to a shared side **141***a*.

On the not-shared side **141***b*, the access command is sent to the destination indicate and global bit addition circuit **142** without being subject to address translation. As a result, address translation is not conducted on the access command sent to the not-shared area.

On the other hand, on the shared side **141***a*, the access command is inputted to the output side translator **144**. FIG. **6** shows details of the output side translator **144**. From an inputted command **141***a*, an access address **1442***a* is separated. It is determined by a range detect circuit **1440** whether the access address **1442***a* is included in the "address in partition" **152** to the "size" **154** of a valid entry of the address translation table. The range detect circuit is provided at the rate of one per entry of the address translation table (**1440** to **1440'**). If the access address **1442***a* is included in the range of address in the partition of the address translation table, then a signal **1440***a* is outputted. If the signal **1440***a* is "1", then the "address in partition" **152** of a pertinent address translation table is outputted to an output A **1441***a*, and the "address in shared memory space" **153** is outputted to an output B **1441***b*. In an address translation circuit **1442**, a new address **1442***b* is calculated according the following equation by using an old base address **1441***a* outputted from the signal A (corresponding to the address in partition in the address translation table), a new base address **1441***b* outputted from the signal B (corresponding to the shared memory area address in the address translation table), and the input address **1442***a*.

New Address=Input Address−Old Base Address+New Base Address

By virtue of this calculation, the input address issued from the inside of the partition (local real address of the partition) can be relocated to an address of the shared memory space. Also in the case where the access address has been included in another entry, similar processing is advanced via the range detect circuit **1440'**. Together with a part of the command other than the address, the output address **1442***b* is sent to the destination indicate and global bit addition circuit **142** via a gate **1445** (enabled by an OR **1443***a* of selection signals). If the input address **1442***a* does not match with any of entries of the address translation table, then an error is detected.

The destination indicate and global bit addition circuit **142** determines which node a command outputted from the inside of the node should be outputted to, and in addition determines the value of the global bit in the command. FIG. **10** shows operation of the destination indicate and global bit addition circuit **142**. The circuit conducts predetermined operation according to the kind of the command and whether the command is a command directed to a not-shared area (in the case where the command is inputted from the signal **141***b* side) or a command directed to a shared area (in the case where the command is inputted from the signal **144***a* side).

(1) Snoop Request (F, FI, I) Command on not-shared Address

After the node configuration in the partition is derived from the partition configuration **160**, the command is multicast to nodes in the partition.

(2) Snoop Request (F, FI, I) Command on Shared Address

The command is broadcasted to all nodes of the system. The global bit is set equal to "1".

(3) Main Memory Access (WB, UW, UR) Command on not-shared Address

On the basis of the main memory configuration information in partition **165**, comparison is conducted to determine which node takes charge of the address range the access address belongs to. The home node (in the local partition) of the access address is derived, and thereafter the command is sent to the home node.

The global bit is set to "0".

(4) Main Memory Access (WB, UW, UR) Command on Shared Address

From the address translation table, the "home node number" 155 of the access address is derived. The command is sent to the home node.

The global bit is set to "1".

(5) Acknowledge (D, DM, ND) Command on not-shared Address

The command is returned to the requesting node. (DND to a not-shared area is not generated.)

The global bit is set to "0".

(6) Acknowledge (D, DM, ND, DND) Command on Shared Address

The command is returned to the requesting node.

The global bit is set to "1".

Owing to the processing heretofore described, the command outputted from the node can be transmitted to suitable destination via the inter-node connection switch **900**.

(B) Processing Conducted when a Node has Received a Command

If a command inputted from the inter-node connection switch **900** is inputted to the address translation circuit **140**, then the command is first inputted to the shared area detect and address check circuit **143**. In the shared area detect and address check circuit **143**, classification of an inputted command and error check are conducted. FIG. **11** shows details of the shared area detect and address check circuit **143**. A command **143*d*** sent from another node is first inputted to a circuit **1430**. The value of the global bit is examined.

If the global bit is "0", i.e., in the case of access to a not-shared area, the command is outputted to a signal **1430*a***. Thereafter, in circuit **1432**, it is checked on the basis of the partition configuration information **160** whether the source node belongs to the local partition. Thereafter, in a circuit **1433**, it is checked on the basis of the main memory configuration information in partition **165** whether the access address is included in main memories of any node in the local partition. A command which has not posed a problem in both checks in the circuits **147** through a signal **143*b***. In this case, address translation is not conducted. A command which has posed a problem in the check of either **1432** or **1433**, i.e., access from the outside of the partition or access to the outside of the main memory in the partition is detected as an error. As a result, unauthorized access to a not-shared area in the partition can be prevented.

Processing conducted in the case where the global bit is "1", i.e., in the case of access to a shared area will now be described. The access command is sent to a circuit **1431** through a signal **1430*b***. In the circuit **1431**, it is checked whether the access address is included in the range of the "address in shared memory space" **153** to the "size" **154** of any of entries of the address translation table **150** (i.e., whether the access address is included in any of shared areas exported or imported by the partition this node belongs to).

If there is the pertinent entry in the address translation table, i.e., if the access address is an address shared by a partition this node belongs to, the command is conveyed to a circuit **1434** through a signal **1431*a***. The DND is returned from a not-shared node. On the other hand, in the case of a command other than the DND, it is checked in the circuit **1434** whether the source node of the access command is included in the bit map **156** of the address translation table representing nodes to be authorized to share the pertinent entry. If there is no problem as a result of the check, the command is sent to the input side translator **145** through a signal **143*a***. In other words, translation from the shared area address to the real address in the partition is conducted.

If there is a problem as a result of the check in the circuit **1434**, i.e., if access is conducted from a node other than partition authorized to share the pertinent shared area (except the DND), then it is detected as an error. As a result, unauthorized access to a shared area in the partition can be prevented.

If there is the pertinent entry in the address translation table, the access address is not included in the shared main memory shared by the partition this node belongs to. This case occurs because in the destination indicate and global bit addition circuit **142** which has transmitted the command, the snoop request command (F, FI, I) on the shared area is broadcasted uniformly to all nodes. Therefore, the command is also sent to nodes which are not included in the sharing nodes. In this case, a DND command indicating that there is no pertinent data in the node must be returned to the access node in order to correctly wait acknowledge to the snoop request. (The access node anticipates acknowledges from all nodes to which the command has been broadcasted.) After being checked in a circuit **1435** that the access command is a snoop request, therefore, the access command is conveyed to the DND acknowledge circuit **146** through a signal **143*c***. (if the access command is not a snoop request, an error is reported.) In the DND acknowledge circuit **146**, the DND command is issued to the source node. The DND command is returned to the accessing node through the destination indicate and global bit addition circuit **142**.

Operation of the input side translator **145** will now be described. In the input side translator, the shared memory space address in the command is translated to a real address in the partition on the basis of information of the address translation table **150**. The input side translator has the same internal configuration as that of the output side translator. The input side translator is different only in the translation direction from the output side translator.

Finally, operation of the acknowledge wait circuit **147** will now be described. The acknowledge wait circuit **147** is a circuit for totalizing acknowledges (D, DM, ND, DND) to F and FI commands which request readout of data from another node. The acknowledge wait circuit **147** does not influence other commands. The acknowledge wait circuit **147** waits until acknowledges (D, DM, ND, or DND) come from all nodes which have issued F or FI commands, i.e., all nodes of the system in the case of shared areas, and all nodes in the partition (indicated by the partition configuration information **160**) in the case of not-shared areas. Then the acknowledge wait circuit **147** conducts the following judgments.

(1) In the case where the D command has been returned:

If the D command has been returned from one node, and ND, DND, or DM has been returned from another node, then the latest data read out by the D command, i.e., data modified in the cache of another node is returned.

(2) In the case where the DM command has been returned:

If the DM command has been returned from one node, and ND or DND has been returned from another node, then data in the main memory read out by the DM command is returned.

(3) In the case where the ND command has been returned:

If ND or DND has been returned from another node, then the own node is the home node. A reply that there are no data is returned. Thereafter, the main memory access circuit **130** reads out data in the main memory **180** and returns the data.

The operation of the acknowledge wait circuit **147** heretofore described is the same as that of the cache coherent check circuit of the conventional hierarchical SMP. Therefore, details of the internal configuration will be omitted.

How access to another node is conducted in the shared memory multiprocessor of the present invention heretofore described will hereafter be described by taking the F command as an example.

(1) F Command on Not-shared Area

In a node which has issued the F command, the F command outputted from the inside of the node is sent from the shared area detect circuit **141** to the destination indicate and global bit addition circuit **142** through the signal **141*b*** (address translation is not conducted), and multicast to nodes in the partition (global bit=0).

In the node which has received the F command, error check is conducted in the shared area detect and address check circuit **143**, and thereafter the F command is conveyed to the inside of the node through the signal **143*b*** and the acknowledge wait circuit **147** (address translation is not conducted).

The acknowledge (D, DM, ND) for the F command is sent from the shared area detect circuit **141** to the destination indicate and global bit addition circuit **142** through the signal **141*b*** (address translation is not conducted), and returned to the accessing node (global bit=0).

In the accessing node which has received the acknowledge for the F command, error check is conducted in the shared area detect and address check circuit **143**, and thereafter the acknowledge is conveyed to the acknowledge wait circuit **147** through the signal **143*b*** (address translation is not conducted). In the acknowledge wait circuit **147**, replies from all nodes in the partition are waited, and are returned to the accessing CPU.

(2) F Command on Shared Area

In the node which has issued the F command, the F command outputted from the inside of the node is sent from the shared area detect circuit **141** to the destination indicate and global bit addition circuit **142** through the output side translator **144**. In this process, the access address is translated from the real address of the accessing partition to the address of the shared memory space. The F command is broadcasted to all nodes of the system (global bit=1).

If the node which has received the F command is included in a partition sharing the accessed shared area, then the F command is subjected to error check in the shared area detect and address check circuit **143**, and thereafter the F command is conveyed to the acknowledge wait circuit **147** through the input side translator. In this process, the access address is translated from the address of the shared memory space to the real address of the accessed partition. Thereafter, the F command is conveyed to the inside of the node. Snoop in the node is conducted by using the local real address of the accessed partition.

The acknowledge (D, DM, ND) for the F command generated as a result of the above described process is sent from the shared area detect circuit **141** to the destination indicate and global bit addition circuit **142** through the output side translator **144**. In this process, the access address is translated from the real address of the accessed partition to the address of the shared memory space again. The command is returned to the accessing node (global bit=1).

If the node which has received the F command is not included in the partition sharing the accessed shared area, then the F command is conveyed to the DND acknowledge circuit **146**, and a DND command is returned to the accessing node (global bit=1).

In the accessing node which has received the acknowledge (D, DM, ND, DND) for the F command, error check is conducted in the shared area detect and address check circuit **143**, and thereafter the acknowledge is conveyed to the acknowledge wait circuit **147** through the input side translator **145**. In this process, the access address is restored from the address of the shared memory space to the real address of the accessing partition. In the acknowledge wait circuit **147**, replies from all nodes of the system are waited, and the replies are returned to the accessing CPU. Answering is conducted by using the local real address of the accessing partition.

As heretofore described, owing to the address translation mechanism of the present invention, data can be accessed by using the local real address in the partition in both the accessing node and the accessed node.

How the system software such as the OS manages the shared memory among partitions of the shared memory multiprocessor in the present invention will now be described. The OS itself may manage the shared memory among partitions, or system software for managing the shared memory among partitions, such as middleware other than the OS, may manage the shared memory among partitions.

In the present system, the partitions do not have a shared memory at all when the system is initialized. Communication between partitions is conducted by using inter-OS communication means (**170** and **910**). Every inter-partition communication for shared memory management hereafter described is conducted by using this inter-OS communication means.

Hereafter, a node having a physical main memory of a shared memory (a node which exports the shared memory is referred to as home node. A partition the home node belongs to is referred to as home partition. Hereafter, with reference to FIGS. **12** to **16**, the procedure of the operation of the system software will be described in detail by dividing it into allocation of a shared area in the main memory, addition of a shared partition, and erasing of a shared area.

As for the management of the shared area, it is hereafter assumed that the home partition basically conducts concentrated management. Also in the case where some unit other than the home partition conducts the management, the management can be realized by using the similar technique.

(A) Allocation of the main memory of the shared area

Allocation of the main memory of the shared area is conducted in the home partition (FIG. **12**). A main memory of a requested size is allocated to be used as a shared area, and necessary data are written into the address translation table. The shared area can be dynamically allocated.

First, an unused entry (an entry having E=0) in the address translation table of each node in the partition is allocated (step **2001**). A real memory of the requested size is allocated in the local main memory (step **2002**). Thereafter, the system software communicates with system software of other partitions, and allocate an area of the requested size in the shared memory space (step **2003**). In all partitions, the shared memory must have the same address map.

Subsequently, the following data concerning the shared area are written into the address translation table of each of nodes in the partition (step **2004**).

"Real address in partition" **152** (start address)

"Address in shared memory space" **153** (start address) "Size of shared area" **154**

"Home node number" **155** (number of node which has secured a main memory)

"Bit map of node to be authorized to share" **156** (initialize it to 0)

At this time point, access authority to the shared area is not given to any partition. Thereafter, "1" is written into E bit of the address translation table of each bode in the partition, and the entry is made valid (step **2005**).

By the processing heretofore described, an area in the main memory can be exported. It becomes possible to access the shared area allocated in the step **2002** from the address in the shared memory space allocated in the step **2003**.

It should be noted here that the same content must be written into address translation tables of all nodes in the partition.

(B) Addition of Sharing Partition

For making it possible for other partitions to access the shared area allocated in (A) (i.e., to import the shared area), the following procedure is necessary. The import processing of the shared area can be dynamically conducted at arbitrary time after the shared area is allocated in the home partition.

(B1) Partition which Newly Imports Shared Area

In a partition of import side which is going to newly share a certain shared area, the following processing is necessary (FIG. 13).

First, an unused entry (an entry having E=0) in the address translation table of each node in the partition is allocated (step **2101**). A real space, to which a physical main memory is not mapped, of a requested size is allocated (step **2102**). This real space is used as a window for accessing a shared area having a physical main memory in another partition. Thereafter, access permission of a pertinent shared area is requested from the system software of the home partition (step **2103**). Here, in a partition which newly imports the shared area, the home node conducts processing of steps **2201** to **2206** (which will be described later) and waits return of a reply of access authorization.

Thereafter, if access authorization comes from the home partition, an entry concerning the shared area of the address translation table sent together with access authorization is received (step **2104**). Contents of the entry of the address translation table received in the step **2104** (except the E bit and address in the partition) are written into the address translation table of each of nodes in the partition, and the real address allocated in step **2102** is written into the "address in partition" field **152** of the address translation table of each of nodes in the partition (step **2105**). Finally, "1" is written into the E bit of the address translation table of each of nodes in the partition.

By the processing heretofore described, import of **10** the shared area is completed, and it is possible to access the shared area in the home node from the real address allocated in the step **2102**.

(B2) Home Partition

In the system software of the partition requested **15** to permit access to the shared area in the step **2103**, processing described hereafter becomes necessary (FIG. **14**). First the access request is checked, and it is decided whether the requesting partition is permitted to share the pertinent shared area (step **2201**). Hereafter, a procedure in the case where the access is permitted will be described.

Subsequently, an old value of the "nodes to be authorized to share" field **156** of an entry corresponding to the shared area of the address translation table is stored into a work variable W (step **2202**). In the "nodes to be authorized to share" field **156** of an entry corresponding to the shared area of the address translation table possessed by each of nodes in the partition, bits corresponding to all nodes which belong to the requesting partition are set (step **2203**). As a result, the requesting partition is authorized to share the pertinent shared area.

Furthermore, the new value of the "nodes to be authorized to share" field **156** is sent to each of partitions (except the own partition) which includes nodes having W set equal to 1, and each of the partitions is requested to modify the field (step **2204**). It should be here noted that the system software which manages shared areas is one in number every partition and each partition needs only to be requested once. Arrival of ACK telling that modification of the "nodes to be authorized to share" field **156** has been completed, from every partition requested in the step **2204** is waited (step **2205**).

Finally, the entry of the shared area of the address translation table is sent to the requesting partition together with the access authorization of the shared area (step **2206**).

By virtue of the processing heretofore described, the "nodes to be authorized to share" field **156** in the address translation table of every node which shares the pertinent shared area is updated, and the requesting partition can share the area.

(B3) Partitions which have Shared the Pertinent Area Until then (Except Home Partition)

In the partitions which have shared the pertinent area until then (except the home partition), the request issued by the home partition in the step **2204** is received, and the "nodes to be authorized to share" field **156** of an entry corresponding to the shared area of the address translation table possessed by each of nodes in the partition is updated. Upon completion of updating, the ACK is returned to the home partition.

(C) Erasing of Shared Area

Erasing of a shared area is started by the home partition, and all partitions which have shared the shared area execute a procedure hereafter described, in cooperation. Erasing of a shared area can also be conducted dynamically. After the shared area has been erased once, resources, such as the resource address translation table, the main memory, and the shared address space, which have been used until then can be utilized again.

It is assumed in the processing hereafter described that the application program has finished the use of the pertinent shared area. Therefore, access to the shared area to be opened is not caused.

(C1) Processing in Home Partition

The home partition makes all other partitions sharing the pertinent area erase the shared area, and thereafter erases the shared area of its own partition.

First, the home partition requests every partition which includes nodes corresponding to 1 in the "nodes to be authorized to share" field **156** of the address translation table (except its own partition) to erase the shared area (step **2301**). And the home partition waits for every requested partition to return the ACK (step **2302**). As a result, it is assured that the shared area has been erased in the partitions which imported the shared area.

Subsequently, the home partition purges all data of the shared area to be erased, in caches of all processors in its own partition (step **2303**). Old data in the cache are thus driven out. Thereafter, the home partition invalidates the entry (i.e., writes "0" in the E bit) corresponding to the shared area to be erased, in the address translation table of every node in the partition (step **2304**). The home partition releases the area in the shared memory space and the physical memory of the shared area (step **2305**). Finally, the home partition purges the TLB of every processor in the partition (step 2306). By virtue of the processing heretofore described, resources used to access the shared area can be completely released.

(C2) Processing in Partitions which Import Shared Area

In each partition requested to erase the shared area by the home partition in the step 2301, processing heretofore described becomes necessary.

First, in all processors in its own partition, data of the shared area to be erased, in caches of all processors in its own partition are purged (step 2401). Thereafter, the pertinent entry in the address translation table of every node in the partition is invalidated (step 2402). The shared memory space, and the area (window area) allocated in the real address space in the partition to access the shared memory are released (step 2403). The TLB of every processor in the partition is purged (step 2404).

As a result, resources used to access the shared area can be completely released. Finally, the ACK is returned to the home partition (step 2405) to inform the home partition that the shared area has been completely released in its own partition.

By virtue of the procedure heretofore described, it is possible in the shared memory multiprocessor of the present invention to dynamically manage the memory shared among partitions.

By virtue of the configuration heretofore described, it is possible in the hierarchical bus connection SMP to realize a memory shared by partitions.

The function of resetting the partitions will now be described in detail. FIG. 17 shows a reset circuit according to the present invention. FIG. 17 shows only node 0. Other nodes also have absolutely the same configuration. Hereafter, the configuration of the node 0 will be described in detail. The CPU, the main memory access circuit, and the address translation circuit in each node are driven by a reset signal 921. Respective nodes have independent reset signals 921 to 928. Respective reset signals are driven by a reset transfer circuit 920. In addition, each node has a reset register 175 which can be accessed from CPUS, and each node can request resetting of other nodes.

First, if an external reset signal (reset at the time of power on, a signal from an ordinary reset button) 929 is driven, then reset signals (921 to 928) of all nodes are made active and all nodes are reset as usual.

The reset register of each node includes a node bit map 1751 indicating nodes to be reset, and an enable bit 1750. If the enable bit 1750 is set to "1", the reset signal is sent to nodes corresponding to "1" in the bit map 1751 through the circuit 920.

By virtue of the circuit heretofore described, it becomes possible for software of each node to reset arbitrary nodes.

In the case where a certain partition (hereafter referred to as partition A) suffers from system down, the following operations are conducted by software of one (hereafter referred to as partition B) of partitions which does not suffer from system down.

(1) The partition B senses that the partition A has suffered from system down. (It can be known by a heart beat function or the like via the shared memory.)

(2) The partition B conducts investigation of error cause and removal of an error cause via the memory shared by partitions, and prepare information so as to allow the partition A to resume its task.

(Since arbitrary addresses including the system area can be shared in the shared memory function of the present invention, error analysis including the OS is possible.)

(3) The partition B sets bits of the bit map 1751 corresponding to nodes of the partition A to "1" (and sets other bits to "0".)

(4) The partition B sets the enable bit 1750 to "1" and then restores it to "0". (As a result, a reset signal is conveyed to respective nodes of the partition A through the reset transfer circuit 920.)

(5) Owing to the processing heretofore described, the partition A can resume the execution. (By using the information of (2), application is resumed.)

By virtue of the reset function heretofore described, it becomes possible for partitions to back up each other, and high reliability can be realized.

The present invention is not limited to the embodiment heretofore described, but can be applied to various variations.

(1) In the foregoing description, the snoop command on the shared area is broadcasted to all nodes in the system.

In contrast thereto, there is also possible such a scheme as to broadcast only to nodes in partitions sharing the accessed shared area in order to reduce the traffic imposed on the switch.

The following points are changed.

When the destination of a command outputted from the inside of a node is decided in the destination indicate and global bit addition circuit 142 (see FIG. 5), the snoop (F, FI, I) command on a shared area is multicast to only nodes corresponding to "1" in the "partitions to be authorized to share" field 156 of the address translation table corresponding to the shared area. (As a result, the snoop command is not sent to partitions which do not share the area.)

The DND command and the DND acknowledge circuit are not used.

When acknowledges of the command on the shared area are waited in the acknowledge wait circuit 147, as many acknowledges as nodes corresponding to "1"s in the "partitions to be authorized to share" field 156 of the address translation table corresponding to the shared area are waited. (As a result, acknowledges of the command in the sharing partitions are waited.)

(2) In the foregoing description, nodes to be authorized to share are stored in the bit map 156 corresponding to each node. However, it is also possible to store nodes to be authorized to share in a bit map corresponding to each partition.

(3) In the foregoing description, the main memory access circuit 130 and the main memory 180 in the node are inserted between the address translation circuit 140 and the CPU bus 190. However, there is also possible such a scheme that the address translation circuit 140 is directly coupled to the CPU bus 190 (but is not directly coupled to the main memory access circuit 130). Apart from the address translation circuit 140, the main memory access circuit 130 is connected to the CPU bus 190. In this case, access from another node to the main memory 180 is conducted via the CPU bus 190. In this case as well, the present invention having the address translation means 140 at the entrance of each node can be utilized as it is.

(4) In the foregoing description, the components 141 to 145 and 147 are shown as separate circuits. By making the circuits of the output side (141,142 and 144) and the circuits of the input side (143, 145 and 147) common respectively, however, duplicated circuits such as the search of the address translation table can be reduced.

(5) In the foregoing description, addresses of the shared memory space can be taken in arbitrary positions. In order to realize arbitrary translation, address translation is realize in a full associative table.

In contrast thereto, by representing the node number by high-order bits of address of the shared memory space and representing the real address in the partition by lower-order bits, the address translation on the export side in the home partition can be significantly reduced. In this case, a real address in the partition can be translated to an address of the shared memory space by simply adding a node number as high-order bits. An address of the shared memory space can be translated to a real address in the partition by simply removing the node number in the high-order bits.

In this case, in order to store, in each node, the management information of shared areas with respect to the whole main memory space in the partition, there is needed a table for storing, every managed unit (having, for example, 1 MB) of shared area, a bit indicating whether sharing is possible (i.e., a bit for storing information as to whether the 1-MB space is a shared area or a not-shared area) and a bit map of nodes to be authorized to share (which is the same information as the field **156** of the address translation table).

As a result, the address translation and access check on the export side (home partition) can be significantly simplified and the hardware can be reduced.

(6) In the foregoing description, the address translation table of the export side is common to the address translation table of the import side. On the import side, however, a table different from that of the export side may be used. In this case, the "home node number" **155** on the export side (which is redundant information) can be eliminated.

(7) In the foregoing description, it is possible to freely read and write data with respect to the shared area. By making the shared area read only from the outside of a partition, however, the hardware can be simplified. In this case, check of the address and the node number (check in the circuits **1432** to **1434**) in the shared area detect and address check circuit **143** is unnecessary. Furthermore, the "nodes to be authorized to share" field **156** needs only to be stored in the home node alone. If a write command such as FI, I, WB or UW comes from the outside of the partition, the shared area detect and address check circuit **143** reports an error.

Furthermore, by combining (5) to (7), the hardware quantity can be significantly reduced.

(8) By adding a read only bit to the address translation table, it is also possible to set only a specific shared area to read only from the outside of the partition. If in that case a write command such as FI, I, WB or UW comes from the outside of the partition to a shared area having a set read only bit, the shared area detect and address check circuit **143** reports an error.

(9) In the foregoing description, the destination indicate and global bit addition circuit **142** located at the exit of each node specifies the destination of the snoop (F, FI, I) command issued from the node. In contrast thereto, there is possible to adopt such a scheme that the inter-node connection switch **900** has information equivalent to the partition configuration information **160** and the destination is specified in the inter-node connection switch **900**. In that case, if the global bit in the command is "1" (i.e., in the case of access to a shared area), the command is broadcasted to all nodes in the system is conducted. If the global bit in the command is "0" (i.e., in the case of access to a not-shared area), the command is multicast only to nodes in the same partition as the destination node.

(10) In the foregoing description, CPUs in the node are connected by the bus **190**. However, other connection forms (such as connection using a switch, or one-to-one connection to the main memory access circuit **130**) may also be used.

(11) In the above described embodiment, the main memory configuration information in partition **165** separately stores the ranges of the main memory respective nodes take charge of, by using pairs of the start address **166** and the end address **167**. However, by using, for example, such a technique as to make the end address of the node n common to the start address of the node n+1, the hardware can be reduced. Furthermore, each node may take charge of a plurality of real address areas by providing a plurality of pairs of the start address **166** and the end address **167** as the areas each node takes charge of.

(12) In the foregoing description, CPUS (**110** to **113**) in the node have independent caches. However, an external cache (level-3 cache) shared by a plurality of CPUs may be provided. It is also possible to provide the main memory access circuit **130** of each node with a copy of the cache TAG of the CPU, and filter the cache coherent transaction coming from another node.

(13) In the foregoing description, the inter-node connection switch **900** is connected by a crossbar network. However, the inter-node connection switch **900** may also be connected by using a network of a different form (such as complete coupling or a multi-stage network).

(14) In the above described embodiment, broadcasting or multicasting is ordered by specifying the destination node in the bit map when issuing a network command to other nodes. However, broadcasting or multicasting may be realized by issuing a plurality of commands for each of the destination nodes from the destination indicate and global bit addition circuit **142**.

According to the present invention, the inside of the hierarchical bus connection SMP is divided into a plurality of partitions. There is provided, at the gateway of each node, means for bidirectionally translating an address of a shared area between an address in a partition and a shared memory space address common to partitions, when realizing a shared memory between partitions. As a result, each partition has a free address space, and data in a shared area can be managed at high speed by a snoop protocol. In addition, an inter-partition shared memory mechanism making possible fault containment between partitions can be realized. Furthermore, by providing means for dynamically generating and erasing the above described address translation information, the shared memory between partitions can be managed flexibly.

What is claimed is:

1. A shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network, said plurality of nodes being divided into a plurality of partitions, each of said partitions including at least one node, the partitions respectively locally sharing a main memory,

wherein a memory address of said system comprises a local real address and an address of common memory space, said local real address being local to each of said plurality of partitions, said address of common memory space being used in common in a memory space used in common between said plurality of partitions,

wherein a main memory of each node is accessed by using a local real address of a partition the node belongs to, and

wherein as for access from each CPU to a main memory of another node, data of a main memory of a node in its partition is accessed via said network by coherently using a local real address of the partition, and access to data of a main memory of a node included in another partition is conducted by translating an access address to an address of said shared memory space when an access command is issued to said network and translating an address of said shared memory space to a local real address of said another partition when said access command enters said node included in said another partition.

2. A shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network, said plurality of nodes being divided into a plurality of partitions each including at least one node locally sharing a main memory, a local real address local to each of said plurality of partitions is used as a memory address of the partition,

wherein said shared memory multiprocessor includes an address translation means provided at a gateway of each of said plurality of nodes to said network, said address translation means being used when a shared area shared by at least two of said partitions is set in any of said main memories, an address specified by a command for accessing said shared area is subjected in said address translation means to mutual translation between said local real address used in a partition and an address of the shared are used in said network, and

wherein a main memory of each node is accessed by using said local real address no matter whether it is said shared area.

3. A shared memory multiprocessor according to claim 2, wherein a command for accessing an area which is not shared between said partitions is exchanged between each node and the network for interconnecting nodes without conducting address translation.

4. A shared memory multiprocessor according to claim 2, wherein when sending out a snoop command from each of said plurality of nodes to the inter-node connection network, a command for accessing an area which is not shared between partitions is multicast to only nodes in the partition, and a command for accessing an area shared by partitions is broadcasted to all nodes belonging to at least partitions sharing the area.

5. A shared memory multiprocessor according to claim 2, wherein in said plurality of partitions, every node included in a partition having said shared area set in a main memory included in its partition has mapping means for mutually mapping between an address in a shared memory space and a local real address of the partition of said shared area.

6. A shared memory multiprocessor according to claim 5, wherein said mapping means conducts mapping in accordance with a table having mapping information set for each shared area, and as a result, a plurality of shared areas can be set as a whole.

7. A shared memory multiprocessor according to claim 2, wherein in said plurality of partitions, on a local real address space of a partition sharing a shared area set in a main memory of another partition, a window area for accessing a shared area set in the main memory of said another partition is provided, said window area does not have a main memory in its own partition, and every node in the partition having said window area has means for mutually mapping between a local real address of the own partition of said window area and said shared memory space address of said shared area.

8. A shared memory multiprocessor according to claim 7, wherein it is made possible for two or more partitions to share one shared area by permitting mapping between

window areas respectively set in a plurality of partitions and a shared memory space address of one shared area.

9. A shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network,

wherein each of said plurality of nodes comprises information addition means, said information addition means being used when said plurality of nodes are divided into a plurality of partitions each including at least one node locally sharing a main memory and a shared area shared by two or more of said partitions, said shared area being provided in any of said main memories, and said information addition means adds information indicating whether a command sent out from a node to said network is a command for accessing said shared area or a command for accessing a local area in said partition which includes said node having said information addition means, to the command,

wherein said local area is addressed by a local address local to said partition and said shared area is addressed by a shared address of said shared area, and

wherein when said command is a command for accessing said shared area a shared address associated with said command is subjected to address translation to a local address to address a main memory in which said shared area is provided.

10. A shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network,

wherein each of said plurality of nodes comprises:

information addition means used when said plurality of nodes are divided into a plurality of partitions, each including at least one node locally sharing a main memory, and a shared memory area shared by two or more of said partitions is provided in any of said main memories, said information addition means adding information, indicating whether a command sent out from that node to said network is a command for accessing said shared area or a command for accessing a local area in said partitions, to the command, and when said command is a command for accessing said shared area, said information addition means translates an address specified by said command from a real address local to a partition including its own node to an address in the shared area and sends out a resultant address into said network, and

means responsive to added information of a command received from said network indicating that the command is a command accessing said shared area, for translating an address of the shared area specified by said command to a real address local to a partition including its own node,

wherein a main memory of each node is accessed by using the real address no matter whether that main memory is said shared area.

11. A shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache

coherent control being conducted between nodes sharing a main memory via said network,

wherein each of said plurality of nodes comprises:

information addition means used when said plurality of nodes are divided into a plurality of partitions, each including at least one node locally sharing a main memory, and a shared area shared by two or more of said partitions is provided in any of said main memories, said information addition means adding information, indicating whether a command sent out from that node to said network is a command for accessing said shared area or a command for accessing a local area in said partitions, to the command,

storage means for setting nodes included in a partition a node belongs to, and

means responsive to a command received from said network being a command for accessing said local area, for checking whether the accessing node of the command is a node included in a partition that node belongs to, and responsive to a negative result, for suppressing access of said command,

wherein said local area is addressed by a local address local to said partition and said shared area is addressed by a shared address of said shared area, and

wherein when said command is a command for accessing said shared area a shared address associated with said command is subjected to address translation to a local address to address a main memory in which said shared area is provided.

12. A shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network, and when said plurality of nodes are divided into a plurality of partitions each including at least one node locally sharing a main memory, a command for locally accessing a main memory in each of said partitions and a command for accessing a shared area of a main memory shared by a plurality of partitions are used,

wherein each of said plurality of nodes comprises:

storage means responsive to determination of partitions sharing said shared area, for setting nodes included in said partitions, and

means responsive to a command received from said network being a command for accessing the local area, for checking whether the accessing node of said command is included in the nodes set in said storage means, and responsive to a negative result, for suppressing access of said command,

wherein said local area is addressed by a local address local to said partition and said shared area is addressed by a shared address of said shared area, and

wherein when said command is a command for accessing said shared area a shared address associated with said command is subjected to address translation to a local address to address a main memory in which said shared area is provided.

13. A shared memory multiprocessor system according to claim **12**, wherein a plurality of said shared areas are set independently, and nodes of a set of partitions authorized to share each of said plurality of shared areas are independently set in said storage means.

14. In a shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network, and when said plurality of nodes are divided into a plurality of partitions each including at least one node locally sharing a main memory, a local real address local to each of said plurality of partitions is used as a memory address of the partition,

a shared area generation method of a shared memory multiprocessor comprising the steps of:

when generating a shared area shared by two or more partitions of said partitions, allocating an area used as a shared area to a main memory of a node in one partition included in partitions attempting to share the shared area, said one partition being own partition; and

defining address mapping to the allocated area and a shared area,

wherein a local area is addressed by a local real address local to said partition and said shared area is addressed by a shared address of said shared area, and

wherein when accessing said shared area a shared address associated with said accessing is subjected to address translation to a local real address to address a main memory in which said shared area is provided.

15. A shared area generation method according to claim **14**, wherein the allocation of an area used as said shared area and the definition of said mapping are carried out by system software prepared in said one partition.

16. In a shared memory multiprocessor system having a plurality of nodes and a network for interconnecting nodes, each of the plurality of nodes including at least one CPU, at least one cache, and at least one main memory, cache coherent control being conducted between nodes sharing a main memory via said network, and when said plurality of nodes are divided into a plurality of partitions each including at least one node locally sharing a main memory, a local real address local to each of said plurality of partitions is used as a memory address of the partition,

a shared area generation method of a shared memory multiprocessor comprising the steps of:

when generating a shared area shared by two or more partitions in any of said partitions, allocating a window area in a partition in a local real address space in the partition, to partitions attempting to share the shared area, other than a partition having an area used as the shared area allocated in a main memory; and

defining address mapping of the allocated area and a shared area,

wherein a local area is addressed by a local real address local to said partition and said shared area is addressed by a shared address of said shared area, and

wherein when accessing said shared area a shared address associated with said accessing is subjected to address translation to a local real address to address a main memory in which said shared area is provided.

17. A shared area generation method according to claim **16**, wherein the allocation of the window area and the definition of mapping are carried out respectively by system software pieces respectively prepared in partitions other than a partition having an area used as a shared area and allocated in a main memory.

* * * * *